Reactive Java with Reactor Internals

Introduction

Spring WebFlux

What is it?

A **non-blocking**, **reactive** web framework that supports Reactive Streams **back pressure**, and runs on servers such as Netty, Undertow, and Servlet 3.1+ containers.

Introduction

WebFlux

Like **WebMVC** but **Reactive**

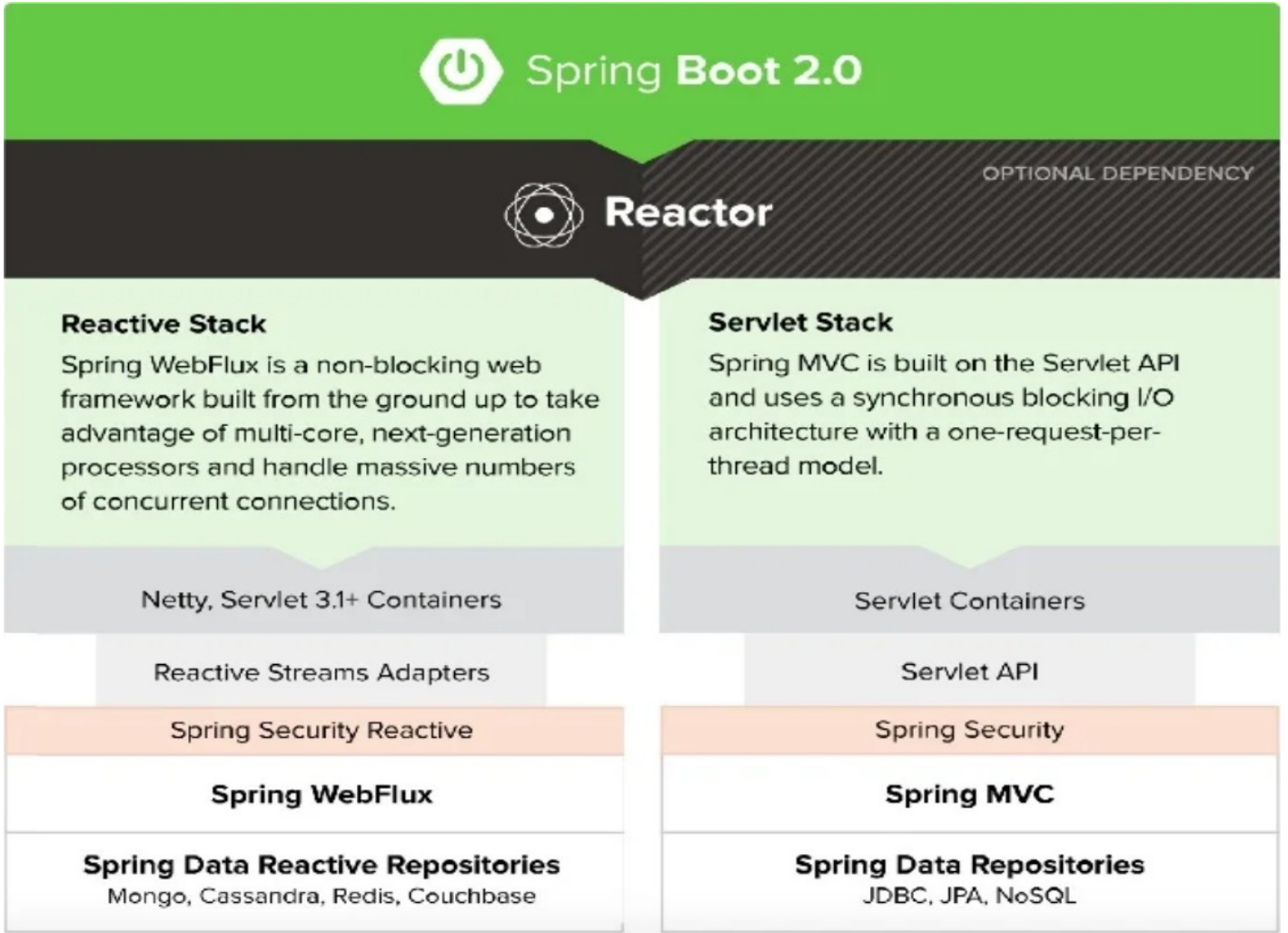
Introduction

Spring WebFlux

Why was it created?

- Because of mobile devices, IoT, and our continuing trend to live online, some apps today have millions of clients.
- Many apps have Black Friday* style usage patterns, where demand can spike exponentially.
- Factors such as these drive the need for a non-blocking web stack that:
  - **handles concurrency with a small number of threads** and
  - **scales with less hardware resources**.

Introduction



Introduction

Spring WebFlux

Other reasons for creating it

- Continuation style APIs enabled by Java 8 lambda expressions allow declarative composition of asynchronous logic
- Lambdas also enabled Spring WebFlux to offer functional web endpoints alongside with annotated controllers

Introduction

Spring WebFlux

What does **reactive** mean?

- **Reactive** refers to programming models (and systems) that are built around asynchronously reacting to external changes (such as messages and events)
- An important mechanism in **reactive** is **non-blocking back pressure** (flow control) *

** In synchronous, imperative code, blocking calls serve as a natural form of back pressure that forces the caller to wait.*

Introduction

Reactive systems vs. Reactive programming

- **Reactive systems** represent an architectural style that allows multiple individual applications to coalesce as a single unit, reacting to its surroundings, while remaining aware of each other
- **Reactive programming** is a subset of asynchronous programming and a paradigm where the availability of new information drives the logic forward rather than having control flow driven by a thread-of-execution

From Reactive programming vs. Reactive systems by Jonas Bonér and Viktor Klang

Introduction

Some Reactive programming use cases

- External Service Calls
- Highly Concurrent Message Consumers
- Spreadsheets
- Abstraction Over (A)synchronous Processing

From: Notes on Reactive Programming Part I: The Reactive Landscape by Dave Syer

Introduction

Reactive Programming in Java

A brief and incomplete history

- Reactive programming ideas have been around for a while, appearing in programming languages (e.g. Erlang) and libraries (e.g. Reactive Extensions for .NET)
- The open source RxJava (Reactive Extensions for Java) project helped move reactive programming forward on the Java platform.
- The Reactive Streams initiative provided a standard and specification for compatibility among reactive implementations in Java. This initiative is a collaboration between engineers from Kaazing, Lightbend, Netflix, Pivotal, Red Hat, Twitter and others.

Introduction

Reactive Programming in Java

- Reactive Streams
- RxJava
- Reactor
- Spring Framework 5
- Ratpack
- Akka
- Vert.x

From: Notes on Reactive Programming Part I: The Reactive Landscape by Dave Syer

Introduction

Reactive Streams:

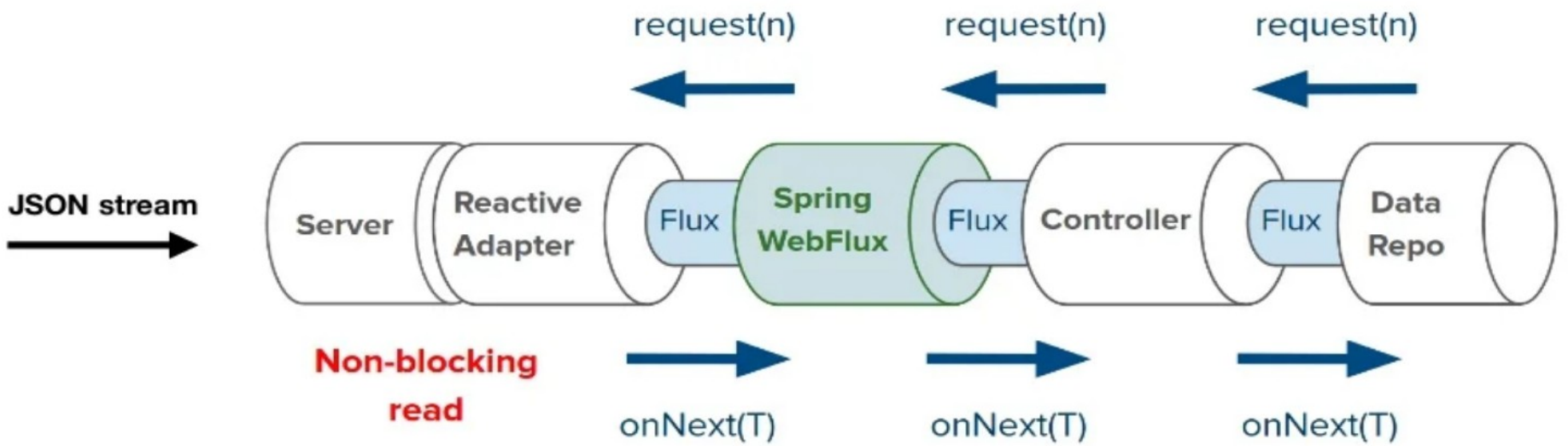
Is a standard and specification for stream-oriented libraries that:

- process a potentially unbounded number of elements
- sequentially,
- with the ability to asynchronously pass elements between components,
- with mandatory non-blocking backpressure.

Introduction

Reactive Streams with Spring

Streaming *to* database with non-blocking back pressure

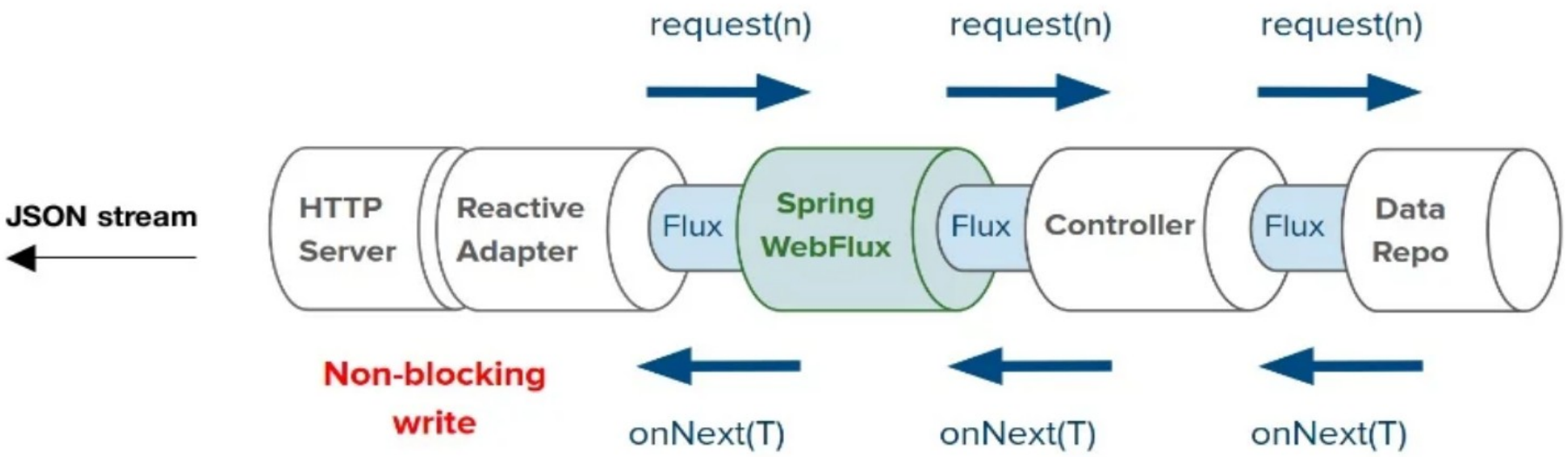


From: Servlet and Reactive Stacks in Spring Framework 5 by Rossen Stoyanchev

Introduction

Reactive Streams with Spring

Streaming *from* database with non-blocking back pressure



From: Servlet and Reactive Stacks in Spring Framework 5 by Rossen Stoyanchev

Introduction

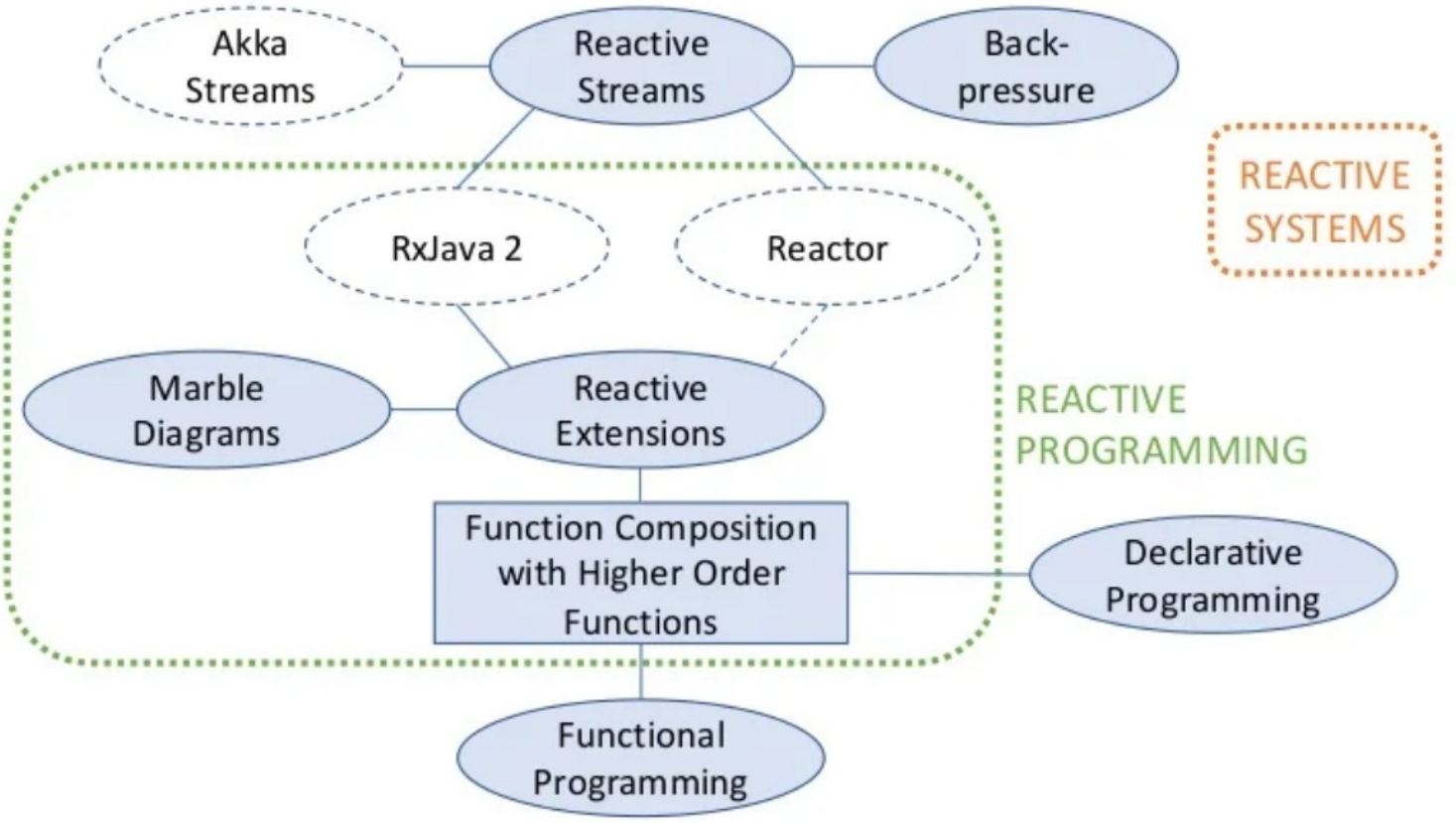
Project Reactor

Avoiding callback hell and other asynchronous pitfalls

Reactive libraries such as Reactor aim to address drawbacks of "classic" asynchronous approaches on the JVM while also focusing on additional aspects:

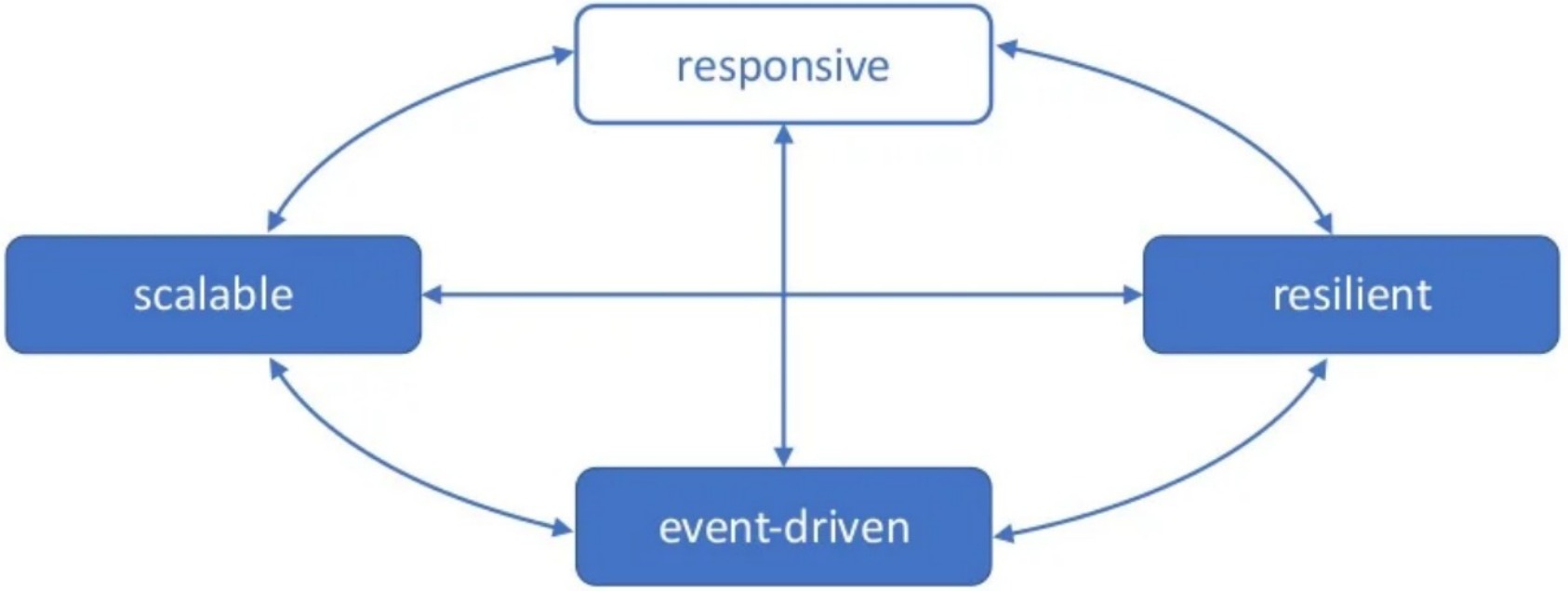
- **Composability** and **readability**
- Data as a **flow** manipulated with a rich vocabulary of **operators**
- Nothing happens until you **subscribe**
- **Backpressure** or the ability for the consumer to signal the producer that the rate of emission is too high
- **High level** but **high value** abstraction that is concurrency-agnostic

Introduction



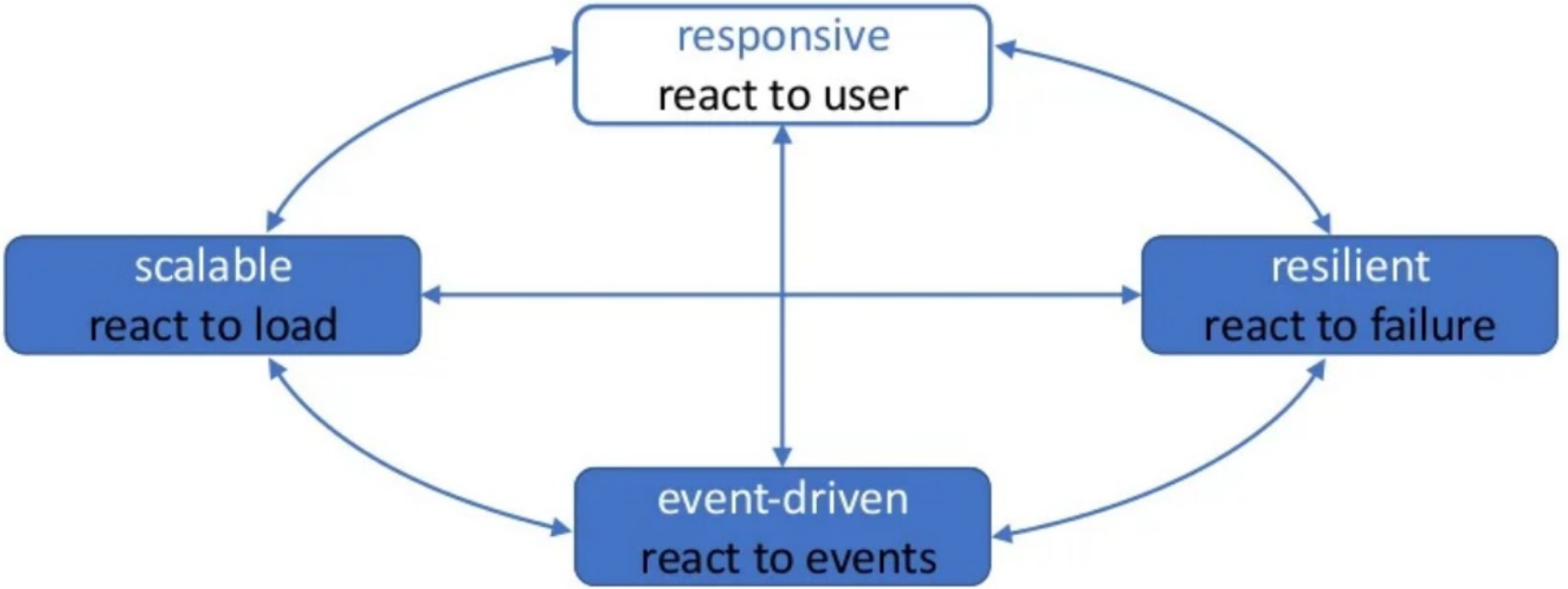
Introduction

The Reactive Manifesto

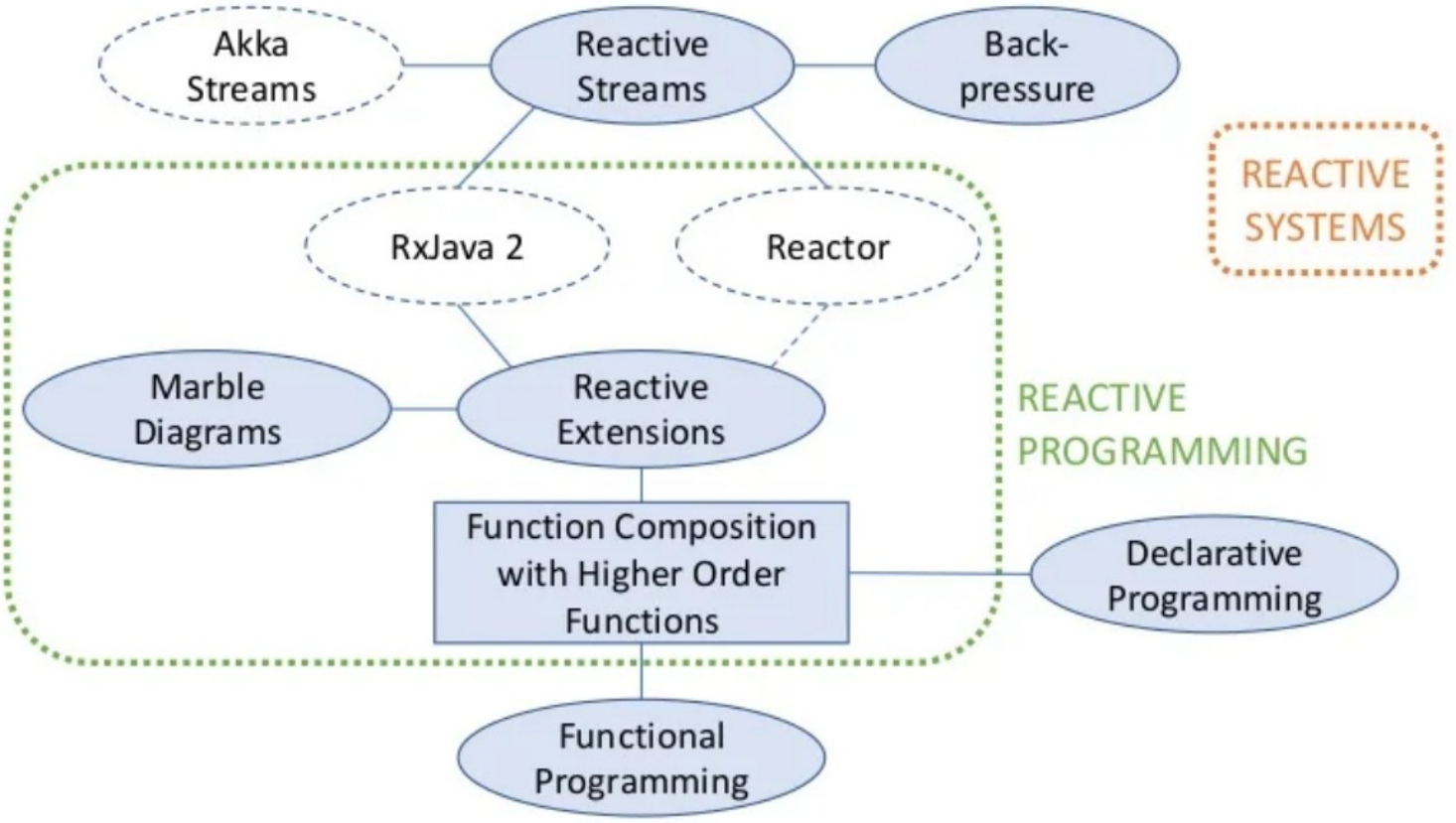


Introduction

The Reactive Manifesto



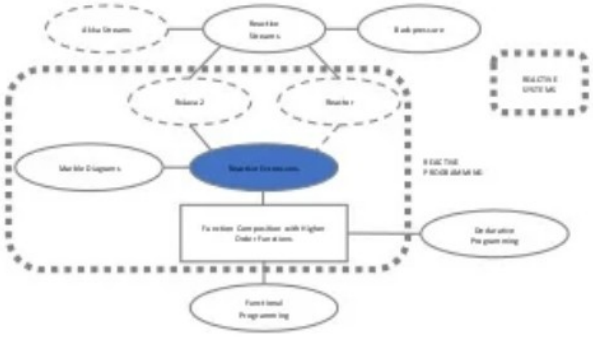
Introduction



Reactive Programming

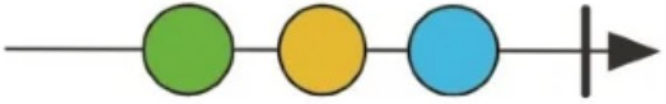
Reactive Extensions

- Observer Pattern
- Operators for transforming, composing, scheduling, error-handling
- Modeling event-based programs
- Avoiding global state and side effects

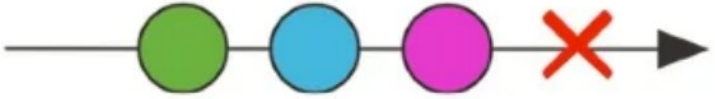


Reactive Programming

Marble Diagrams



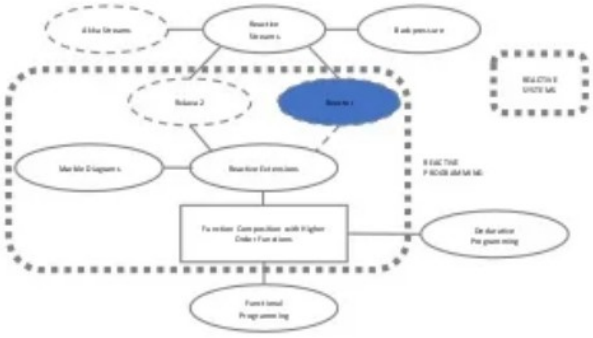
Stream emitting 3 events, followed by completion.



Stream emitting 3 events, completed by error.

Reactive Programming

Reactor



```
Flux.create(emitter -> {
  emitter.next(1);
  emitter.next(2);
  emitter.next(3);
  emitter.complete();
});
```

Stream emitting 3 events, followed by completion.

Reactive Programming

Reactor

```
Flux.create(emitter -> {
  emitter.next(1);
  emitter.next(2);
  emitter.next(3);
  emitter.complete();
});
```

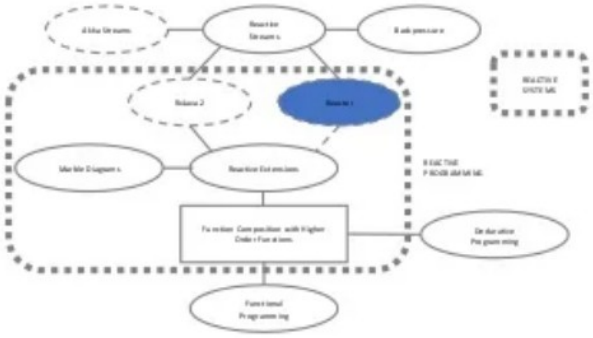
Stream emitting 3 events, followed by completion.

```
Flux.create(emitter -> {
  emitter.next(1);
  emitter.next(2);
  emitter.next(3);
  emitter.error(new RuntimeException());
});
```

Stream emitting 3 events, completed by error.

Reactive Programming

Subscribe



```
log.info("Before Flux.create()");
Flux<Integer> flux = Flux.create(emitter -> {
  log.info("emitter.next({})", 1);
  emitter.next(1);
  log.info("emitter.complete()");
  emitter.complete();
});
log.info("After Flux.create()");
```

Reactive Programming

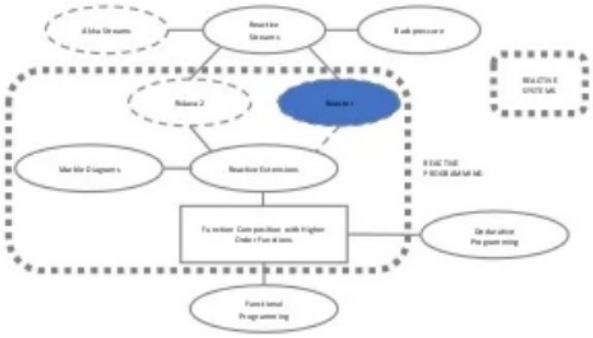
Subscribe

```
log.info("Before Flux.create()");
Flux<Integer> flux = Flux.create(emitter -> {
  log.info("emitter.next({})", 1);
  emitter.next(1);
  log.info("emitter.complete()");
  emitter.complete();
});
log.info("After Flux.create()");
```

```
[main] - Before Flux.create()
```

Reactive Programming

Subscribe



```
log.info("Before Flux.create()");
Flux<Integer> flux = Flux.create(emitter -> {
  log.info("emitter.next({})", 1);
  emitter.next(1);
  log.info("emitter.complete()");
  emitter.complete();
});
log.info("After Flux.create()");
```

```
[main] - Before Flux.create()
[main] - After Flux.create()
```

Reactive Programming

Subscribe

```
log.info("Before Flux.create()");
Flux<Integer> flux = Flux.create(emitter -> {
  log.info("emitter.next({})", 1);
  emitter.next(1);
  log.info("emitter.complete()");
  emitter.complete();
});
log.info("After Flux.create()");

log.info("Before Flux.subscribe()");
flux.subscribe(
  next -> log.info("subscriber.onNext({})", next),
  error -> log.info("subscriber.onError({})", error),
  () -> log.info("subscriber.onComplete()")
);
log.info("After Flux.subscribe()");
```

Reactive Programming

Subscribe

```
log.info("Before Flux.create()");
Flux<Integer> flux = Flux.create(emitter -> {
  log.info("emitter.next({})", 1);
  emitter.next(1);
  log.info("emitter.complete()");
  emitter.complete();
});
log.info("After Flux.create()");

log.info("Before Flux.subscribe()");
flux.subscribe(
  next -> log.info("subscriber.onNext({})", next),
  error -> log.info("subscriber.onError({})", error),
  () -> log.info("subscriber.onComplete()")
);
log.info("After Flux.subscribe()");
```

```
[main] - Before Flux.create()
```

Reactive Programming

Subscribe

```
log.info("Before Flux.create()");
Flux<Integer> flux = Flux.create(emitter -> {
  log.info("emitter.next({})", 1);
  emitter.next(1);
  log.info("emitter.complete()");
  emitter.complete();
});
log.info("After Flux.create()");

log.info("Before Flux.subscribe()");
flux.subscribe(
  next -> log.info("subscriber.onNext({})", next),
  error -> log.info("subscriber.onError({})", error),
  () -> log.info("subscriber.onComplete()")
);
log.info("After Flux.subscribe()");
```

```
[main] - Before Flux.create()
[main] - After Flux.create()
```

Reactive Programming

Subscribe

```
log.info("Before Flux.create()");
Flux<Integer> flux = Flux.create(emitter -> {
  log.info("emitter.next({})", 1);
  emitter.next(1);
  log.info("emitter.complete()");
  emitter.complete();
});
log.info("After Flux.create()");

log.info("Before Flux.subscribe()");
flux.subscribe(
  next -> log.info("subscriber.onNext({})", next),
  error -> log.info("subscriber.onError({})", error),
  () -> log.info("subscriber.onComplete()")
);
log.info("After Flux.subscribe()");
```

```
[main] - Before Flux.create()
[main] - After Flux.create()
[main] - Before Flux.subscribe()
```

Reactive Programming

Subscribe

```
log.info("Before Flux.create()");
Flux<Integer> flux = Flux.create(emitter -> {
  log.info("emitter.next({})", 1);
  emitter.next(1);
  log.info("emitter.complete()");
  emitter.complete();
});
log.info("After Flux.create()");

log.info("Before Flux.subscribe()");
flux.subscribe(
  next -> log.info("subscriber.onNext({})", next),
  error -> log.info("subscriber.onError({})", error),
  () -> log.info("subscriber.onComplete()")
);
log.info("After Flux.subscribe()");
```

```
[main] - Before Flux.create()
[main] - After Flux.create()
[main] - Before Flux.subscribe()
[main] - emitter.next(1)
```

Reactive Programming

Subscribe

```
log.info("Before Flux.create()");
Flux<Integer> flux = Flux.create(emitter -> {
  log.info("emitter.next({})", 1);
  emitter.next(1);
  log.info("emitter.complete()");
  emitter.complete();
});
log.info("After Flux.create()");

log.info("Before Flux.subscribe()");
flux.subscribe(
  next -> log.info("subscriber.onNext({})", next),
  error -> log.info("subscriber.onError({})", error),
  () -> log.info("subscriber.onComplete()")
);
log.info("After Flux.subscribe()");
```

```
[main] - Before Flux.create()
[main] - After Flux.create()
[main] - Before Flux.subscribe()
[main] - emitter.next(1)
[main] - subscriber.onNext(1)
```

Reactive Programming

Subscribe

```
log.info("Before Flux.create()");
Flux<Integer> flux = Flux.create(emitter -> {
  log.info("emitter.next({})", 1);
  emitter.next(1);
  log.info("emitter.complete()");
  emitter.complete();
});
log.info("After Flux.create()");

log.info("Before Flux.subscribe()");
flux.subscribe(
  next -> log.info("subscriber.onNext({})", next),
  error -> log.info("subscriber.onError({})", error),
  () -> log.info("subscriber.onComplete()")
);
log.info("After Flux.subscribe()");
```

```
[main] - Before Flux.create()
[main] - After Flux.create()
[main] - Before Flux.subscribe()
[main] - emitter.next(1)
[main] - subscriber.onNext(1)
[main] - emitter.complete()
```

Reactive Programming

Subscribe

```
log.info("Before Flux.create()");
Flux<Integer> flux = Flux.create(emitter -> {
  log.info("emitter.next({})", 1);
  emitter.next(1);
  log.info("emitter.complete()");
  emitter.complete();
});
log.info("After Flux.create()");

log.info("Before Flux.subscribe()");
flux.subscribe(
  next -> log.info("subscriber.onNext({})", next),
  error -> log.info("subscriber.onError({})", error),
  () -> log.info("subscriber.onComplete()")
);
log.info("After Flux.subscribe()");
```

```
[main] - Before Flux.create()
[main] - After Flux.create()
[main] - Before Flux.subscribe()
[main] - emitter.next(1)
[main] - subscriber.onNext(1)
[main] - emitter.complete()
[main] - subscriber.onComplete()
```

Reactive Programming

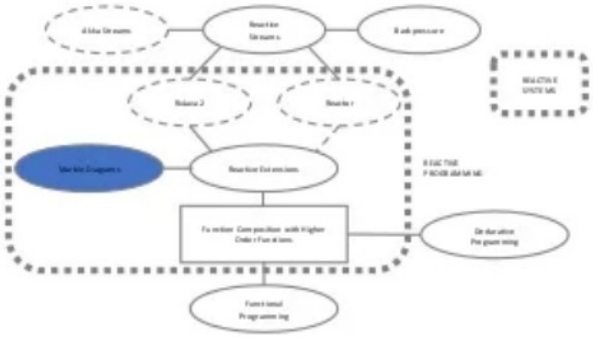
Subscribe

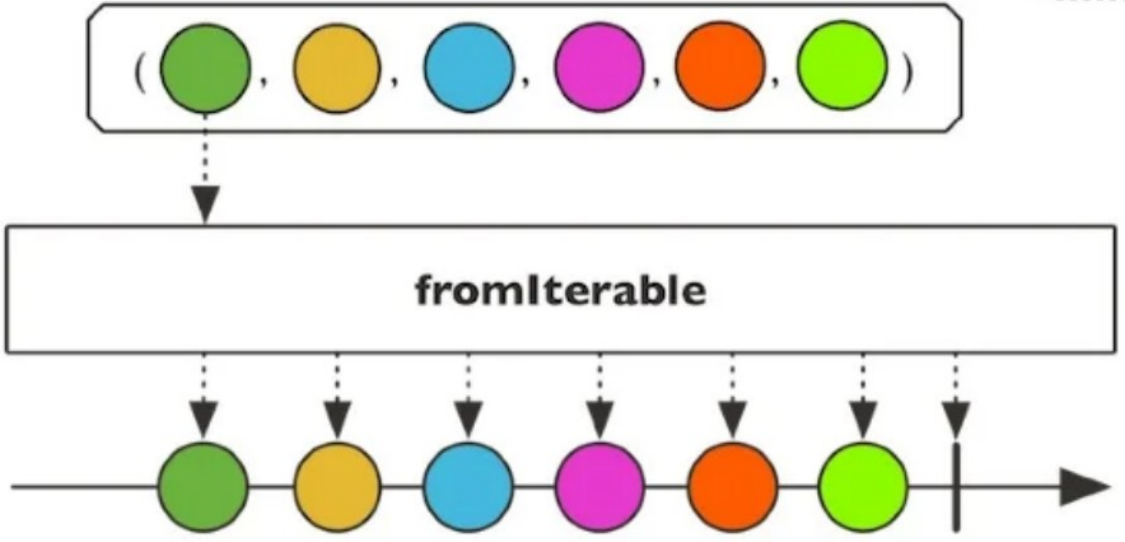
```
log.info("Before Flux.create()");
Flux<Integer> flux = Flux.create(emitter -> {
  log.info("emitter.next({})", 1);
  emitter.next(1);
  log.info("emitter.complete()");
  emitter.complete();
});
log.info("After Flux.create()");

log.info("Before Flux.subscribe()");
flux.subscribe(
  next -> log.info("subscriber.onNext({})", next),
  error -> log.info("subscriber.onError({})", error),
  () -> log.info("subscriber.onComplete()")
);
log.info("After Flux.subscribe()");
```

```
[main] - Before Flux.create()
[main] - After Flux.create()
[main] - Before Flux.subscribe()
[main] - emitter.next(1)
[main] - subscriber.onNext(1)
[main] - emitter.complete()
[main] - subscriber.onComplete()
[main] - After Flux.subscribe()
```

Reactive Programming

Factory methods: fromIterable

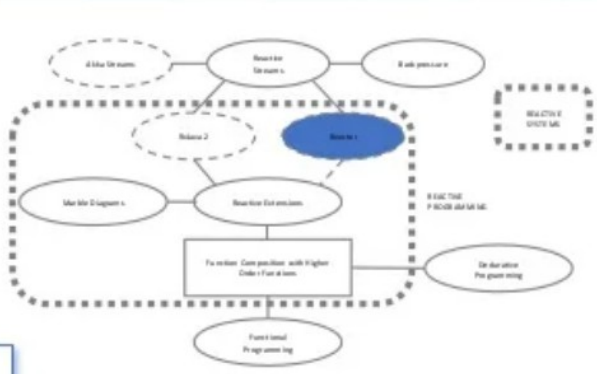




https://raw.githubusercontent.com/reactor/reactor-core/v3.0.6.RELEASE/src/docs/marble/fromiterable.png

Reactive Programming

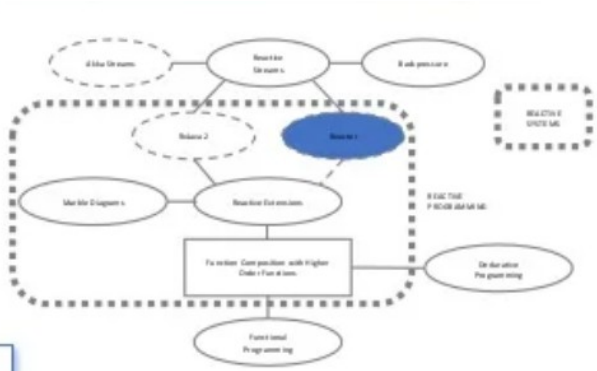
Factory methods: fromIterable



```
public <T> Flux<T> fromIterable(Iterable<T> iterable) {
  return Flux.create(emitter -> {
    Iterator<T> iterator = iterable.iterator();

    while (iterator.hasNext()) {
      emitter.next(iterator.next());
    }

    emitter.complete();
  });
}
```

Reactive Programming

Factory methods: fromIterable



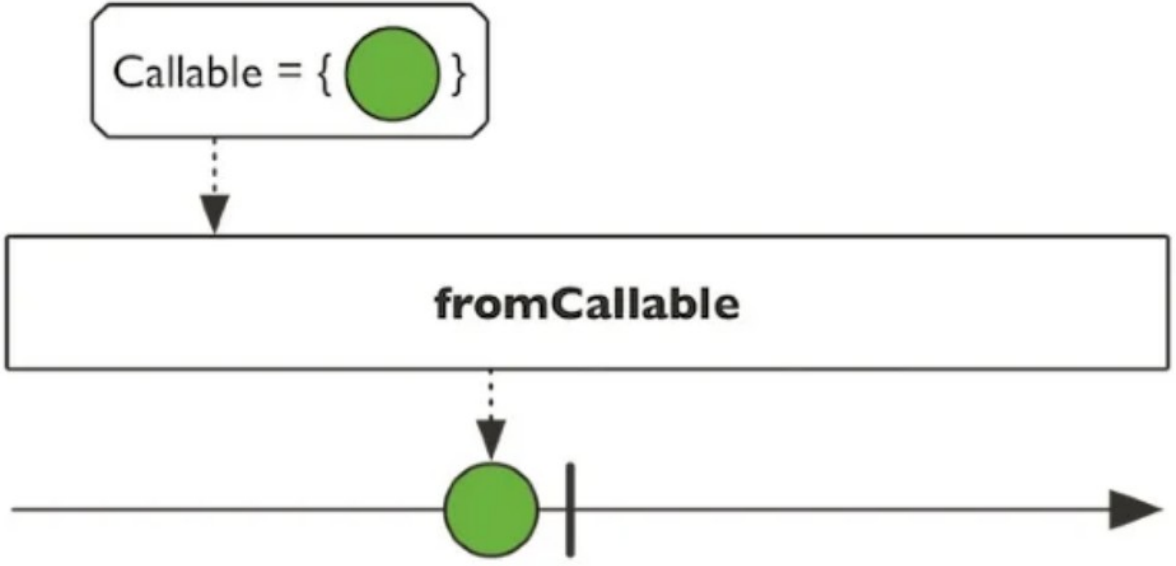
```
public <T> Flux<T> fromIterable(Iterable<T> iterable) {
  return Flux.create(emitter -> {
    Iterator<T> iterator = iterable.iterator();

    while (iterator.hasNext()) {
      emitter.next(iterator.next());
    }

    emitter.complete();
  });
}
```

naive implementation

Reactive Programming

Factory methods: fromCallable

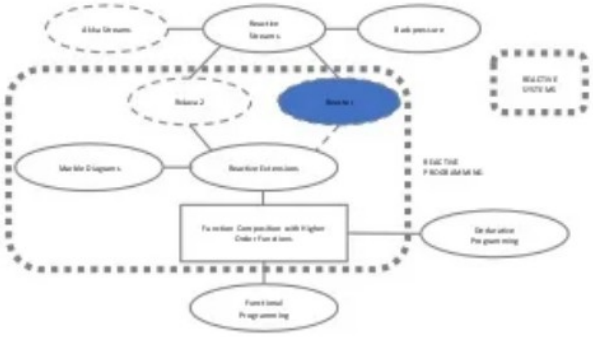
```
Mono<User> maybeUser = Mono
    .fromCallable(() -> loadUser(cwid))
```



https://raw.githubusercontent.com/reactor/reactor-core/v3.0.6.RELEASE/src/docs/marble/fromcallable.png

Reactive Programming

StepVerifier



```
Flux<String> flux = Flux.fromIterable(cwids).take(5);
```

Reactive Programming

StepVerifier

```
Flux<String> flux = Flux.fromIterable(cwids).take(5);

StepVerifier.create(flux)
```

Reactive Programming

StepVerifier

```
Flux<String> flux = Flux.fromIterable(cwids).take(5);

StepVerifier.create(flux)
  .expectSubscription()
```

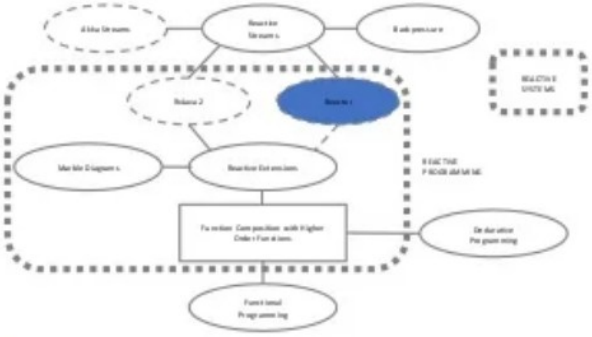
Reactive Programming

StepVerifier

```
Flux<String> flux = Flux.fromIterable(cwids).take(5);

StepVerifier.create(flux)
  .expectSubscription()
  .expectNext(cwids.get(0))
```

Reactive Programming

StepVerifier



```
Flux<String> flux = Flux.fromIterable(cwids).take(5);

StepVerifier.create(flux)
  .expectSubscription()
  .expectNext(cwids.get(0))
  .expectNext(cwids.get(1))
```

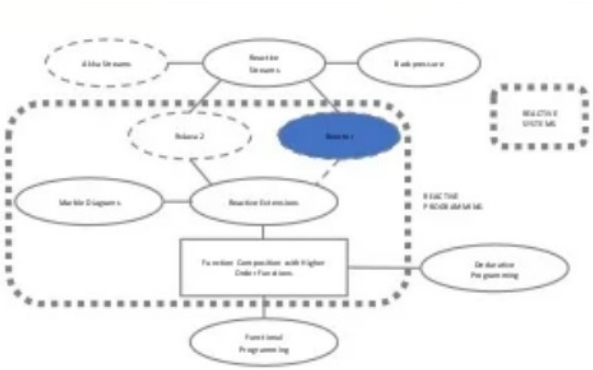
Reactive Programming

StepVerifier

```
Flux<String> flux = Flux.fromIterable(cwids).take(5);

StepVerifier.create(flux)
  .expectSubscription()
  .expectNext(cwids.get(0))
  .expectNext(cwids.get(1))
  .expectNext(cwids.get(2), cwids.get(3), cwids.get(4))
```

Reactive Programming

StepVerifier



```
Flux<String> flux = Flux.fromIterable(cwids).take(5);

StepVerifier.create(flux)
  .expectSubscription()
  .expectNext(cwids.get(0))
  .expectNext(cwids.get(1))
  .expectNext(cwids.get(2), cwids.get(3), cwids.get(4))
  .expectComplete()
```

Reactive Programming

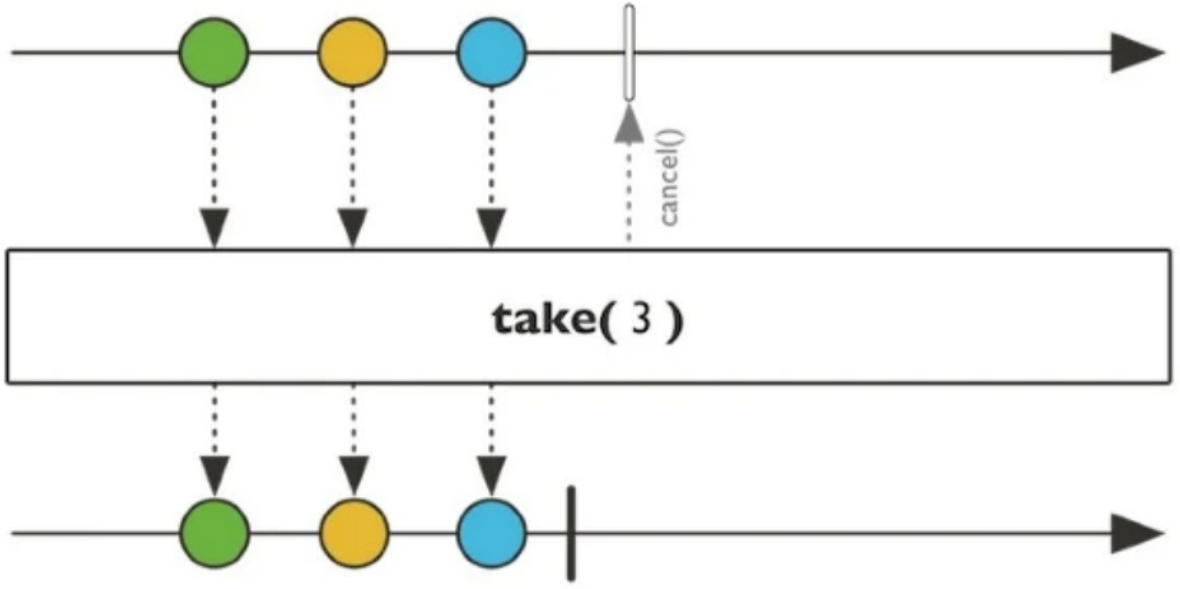
StepVerifier

```
Flux<String> flux = Flux.fromIterable(cwids).take(5);

StepVerifier.create(flux)
  .expectSubscription()
  .expectNext(cwids.get(0))
  .expectNext(cwids.get(1))
  .expectNext(cwids.get(2), cwids.get(3), cwids.get(4))
  .expectComplete()
  .verify();
```

Reactive Programming

Operators: take

```
Flux.fromIterable(cwids)
  .take(3)
  ...
```

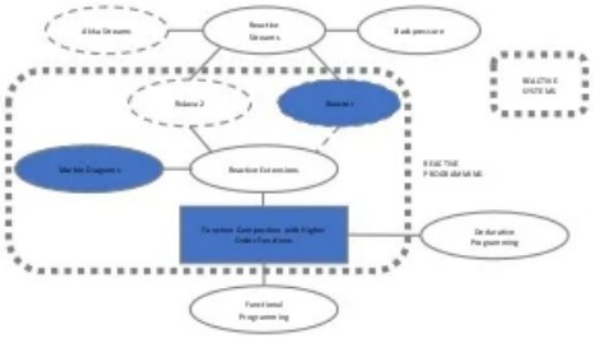


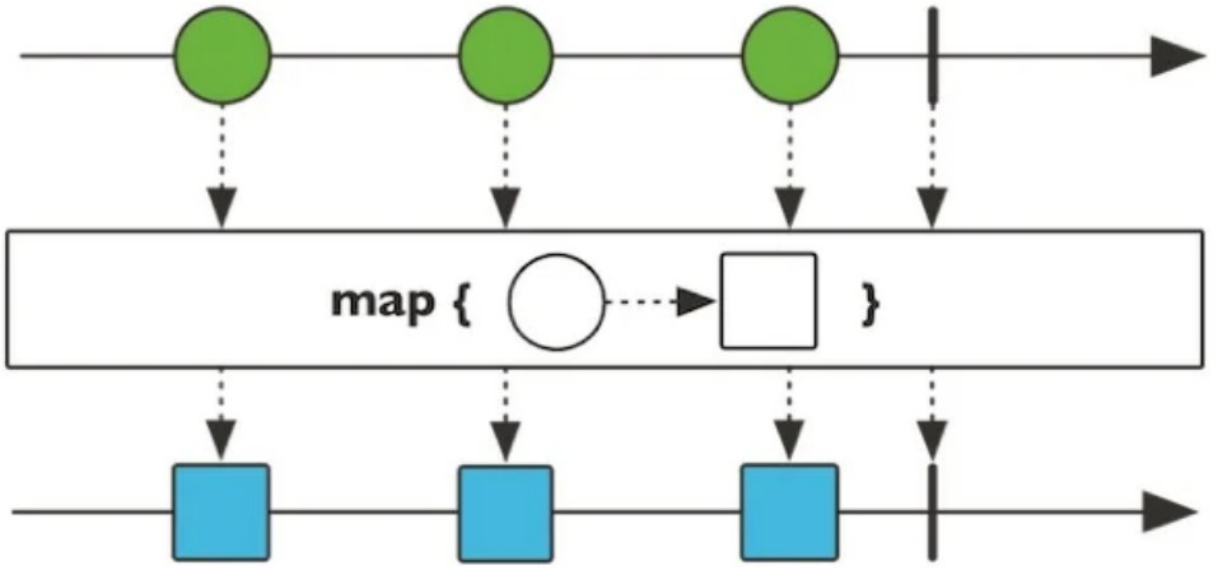
https://raw.githubusercontent.com/reactor/reactor-core/master/src/docs/marble/take.png

Reactive Programming

Operators: map

```
Flux.fromIterable(cwids)
  .map(cwid -> loadUser(cwid))
  ...
```



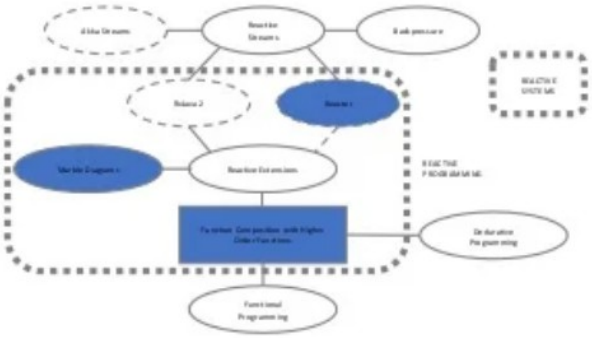


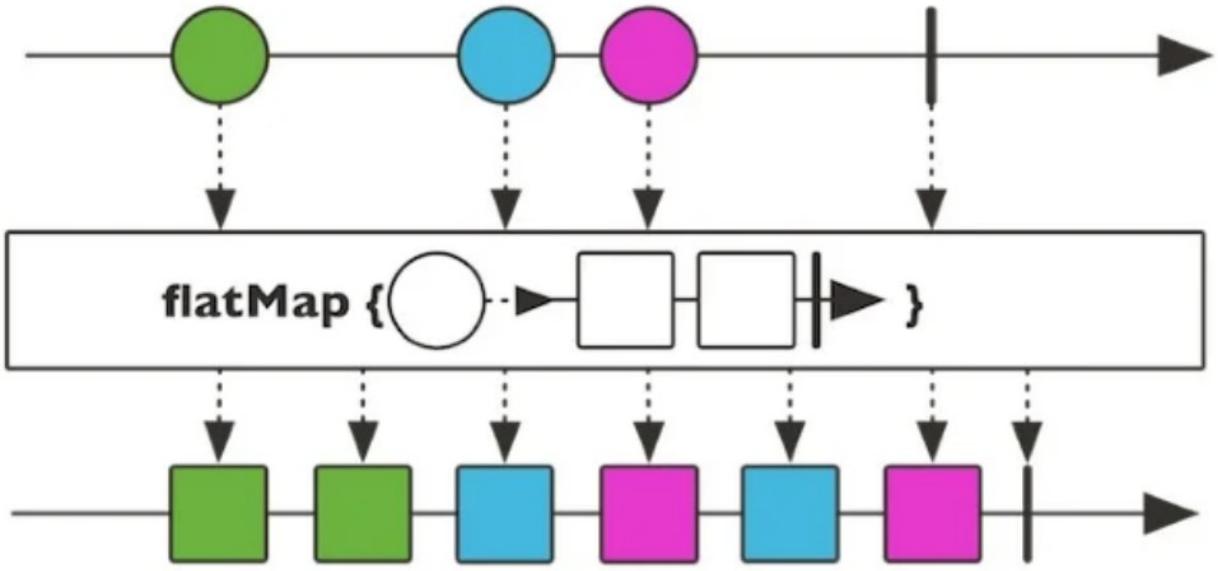
https://raw.githubusercontent.com/reactor/reactor-core/master/src/docs/marble/map.png

Reactive Programming

Operators: flatMap

```
Flux.fromIterable(cwids)
  .flatMap(cwid -> loadUserAsMono(cwid))
  ...
```

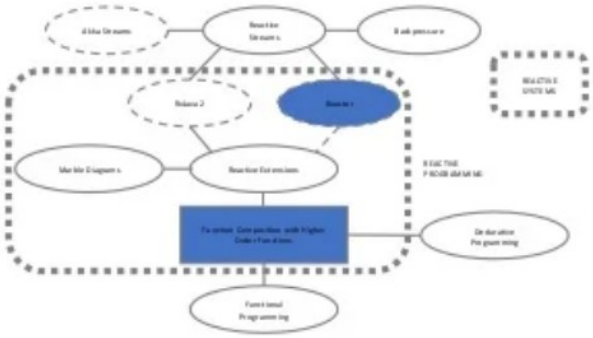




https://raw.githubusercontent.com/reactor/reactor-core/master/src/docs/marble/flatmap.png

Reactive Programming

Operators: flatMap



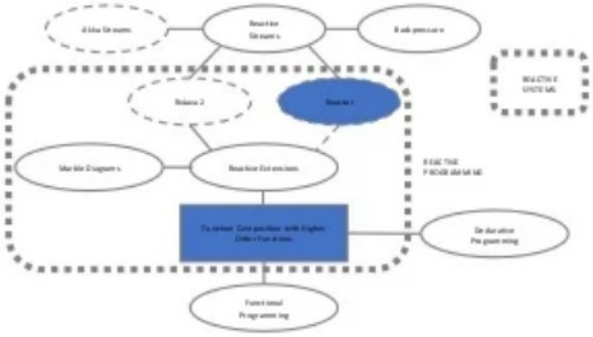
```
Flux<User> users = Flux
  .fromIterable(cwids)
  .flatMap(cwid -> loadUserAsMono(cwid));
```

Reactive Programming

Operators: flatMap

```
private Mono<User> loadUserAsMono(String cwid) {
  return Mono.fromCallable(() -> loadUser(cwid));
}
```

```
Flux<User> users = Flux
  .fromIterable(cwids)
  .flatMap(cwid -> loadUserAsMono(cwid));
```

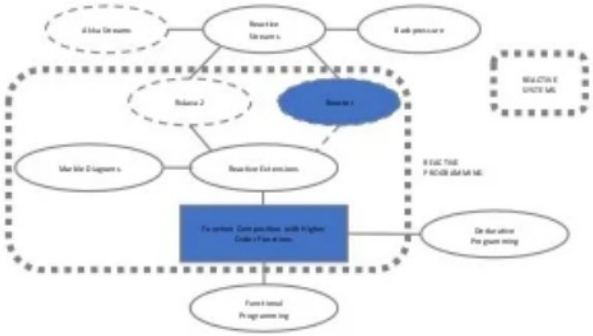


Reactive Programming

Operators: flatMap

```
private Mono<User> loadUserAsMono(String cwid) {
  return Mono.fromCallable(() -> loadUser(cwid));
}
```

```
Flux<User> users = Flux
  .fromIterable(cwids)
  .flatMap(cwid -> loadUserAsMono(cwid));

StepVerifier.create(users)
```

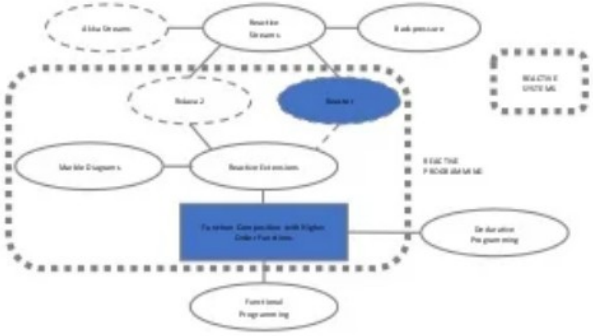


Reactive Programming

Operators: flatMap

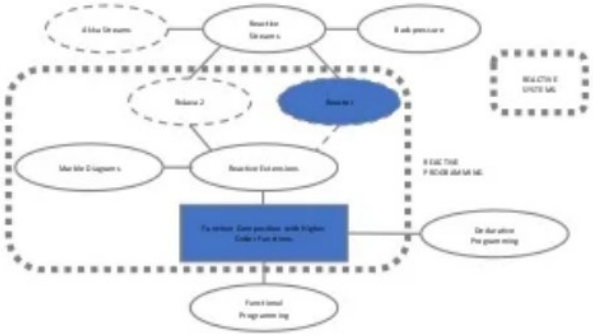
```
private Mono<User> loadUserAsMono(String cwid) {
  return Mono.fromCallable(() -> loadUser(cwid));
}
```

```
Flux<User> users = Flux
  .fromIterable(cwids)
  .flatMap(cwid -> loadUserAsMono(cwid));

StepVerifier.create(users)
  .expectNextMatches(hasCwid(cwids.get(0)))
```



Reactive Programming

Operators: flatMap

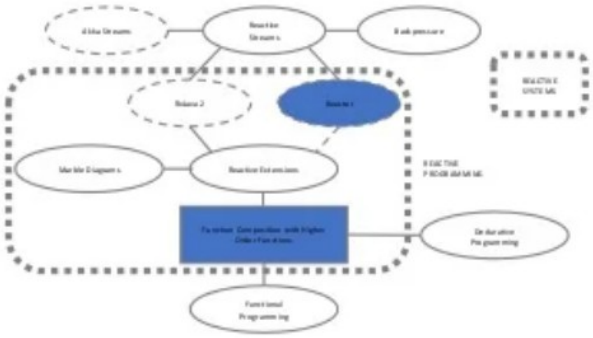


```
private Mono<User> loadUserAsMono(String cwid) {
  return Mono.fromCallable(() -> loadUser(cwid));
}
```

```
Flux<User> users = Flux
  .fromIterable(cwids)
  .flatMap(cwid -> loadUserAsMono(cwid));

StepVerifier.create(users)
  .expectNextMatches(hasCwid(cwids.get(0)))
  .expectNextMatches(hasCwid(cwids.get(1)))
```

Reactive Programming

Operators: flatMap



```
private Mono<User> loadUserAsMono(String cwid) {
  return Mono.fromCallable(() -> loadUser(cwid));
}
```

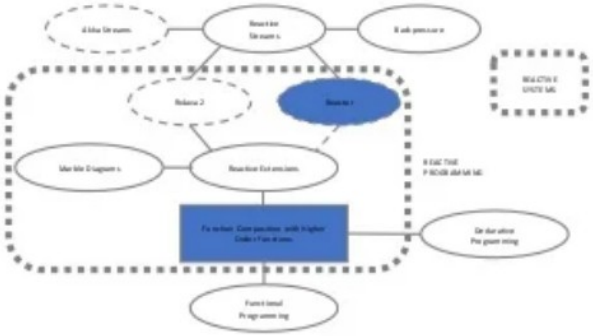
```
Flux<User> users = Flux
  .fromIterable(cwids)
  .flatMap(cwid -> loadUserAsMono(cwid));

StepVerifier.create(users)
  .expectNextMatches(hasCwid(cwids.get(0)))
  .expectNextMatches(hasCwid(cwids.get(1)))
  .expectNextMatches(hasCwid(cwids.get(2)))
```

Reactive Programming

Operators: flatMap

```
private Mono<User> loadUserAsMono(String cwid) {
  return Mono.fromCallable(() -> loadUser(cwid));
}
```

```
Flux<User> users = Flux
  .fromIterable(cwids)
  .flatMap(cwid -> loadUserAsMono(cwid));

StepVerifier.create(users)
  .expectNextMatches(hasCwid(cwids.get(0)))
  .expectNextMatches(hasCwid(cwids.get(1)))
  .expectNextMatches(hasCwid(cwids.get(2)))
  ...
  .verifyComplete();
```



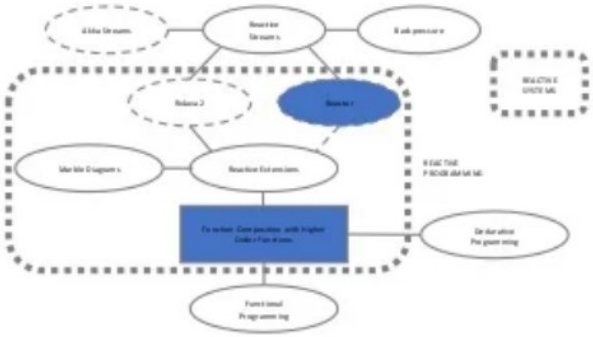
Reactive Programming

Operators: flatMap

```
private Mono<User> loadUserAsMono(String cwid) {
  return Mono.fromCallable(() -> loadUser(cwid));
}
```

```
Flux<User> users = Flux
  .fromIterable(cwids)
  .flatMap(cwid -> loadUserAsMono(cwid));

StepVerifier.create(users)
  .expectNextMatches(hasCwid(cwids.get(0)))
  .expectNextMatches(hasCwid(cwids.get(1)))
  .expectNextMatches(hasCwid(cwids.get(2)))
  ...
  .verifyComplete();
```

?



Reactive Programming

SubscribeOn

```
log.info("Before Flux.create()");
Flux<Integer> flux = Flux.create(emitter -> {
  log.info("emitter.next({})", 1);
  emitter.next(1);
  log.info("emitter.complete()");
  emitter.complete();
})                                    ;
log.info("After Flux.create()");

log.info("Before Flux.subscribe()");
flux.subscribe(
  next -> log.info("subscriber.onNext({})", next),
  error -> log.info("subscriber.onError({})", error),
  () -> log.info("subscriber.onComplete()")
);
log.info("After Flux.subscribe()");
```

Reactive Programming

SubscribeOn

```
log.info("Before Flux.create()");
Flux<Integer> flux = Flux.create(emitter -> {
  log.info("emitter.next({})", 1);
  emitter.next(1);
  log.info("emitter.complete()");
  emitter.complete();
}).subscribeOn(Schedulers.parallel());
log.info("After Flux.create()");

log.info("Before Flux.subscribe()");
flux.subscribe(
  next -> log.info("subscriber.onNext({})", next),
  error -> log.info("subscriber.onError({})", error),
  () -> log.info("subscriber.onComplete()")
);
log.info("After Flux.subscribe()");
```

Reactive Programming

SubscribeOn

```
log.info("Before Flux.create()");
Flux<Integer> flux = Flux.create(emitter -> {
  log.info("emitter.next({})", 1);
  emitter.next(1);
  log.info("emitter.complete()");
  emitter.complete();
}).subscribeOn(Schedulers.parallel());
log.info("After Flux.create()");

log.info("Before Flux.subscribe()");
flux.subscribe(
  next -> log.info("subscriber.onNext({})", next),
  error -> log.info("subscriber.onError({})", error),
  () -> log.info("subscriber.onComplete()")
);
log.info("After Flux.subscribe()");
```

```
[main] - Before Flux.create()
```

Reactive Programming

SubscribeOn

```
log.info("Before Flux.create()");
Flux<Integer> flux = Flux.create(emitter -> {
  log.info("emitter.next({})", 1);
  emitter.next(1);
  log.info("emitter.complete()");
  emitter.complete();
}).subscribeOn(Schedulers.parallel());
log.info("After Flux.create()");

log.info("Before Flux.subscribe()");
flux.subscribe(
  next -> log.info("subscriber.onNext({})", next),
  error -> log.info("subscriber.onError({})", error),
  () -> log.info("subscriber.onComplete()")
);
log.info("After Flux.subscribe()");
```

```
[main] - Before Flux.create()
[main] - After Flux.create()
```

Reactive Programming

SubscribeOn

```
log.info("Before Flux.create()");
Flux<Integer> flux = Flux.create(emitter -> {
  log.info("emitter.next({})", 1);
  emitter.next(1);
  log.info("emitter.complete()");
  emitter.complete();
}).subscribeOn(Schedulers.parallel());
log.info("After Flux.create()");

log.info("Before Flux.subscribe()");
flux.subscribe(
  next -> log.info("subscriber.onNext({})", next),
  error -> log.info("subscriber.onError({})", error),
  () -> log.info("subscriber.onComplete()")
);
log.info("After Flux.subscribe()");
```

```
[main] - Before Flux.create()
[main] - After Flux.create()
[main] - Before Flux.subscribe()
```

Reactive Programming

SubscribeOn

```
log.info("Before Flux.create()");
Flux<Integer> flux = Flux.create(emitter -> {
  log.info("emitter.next({})", 1);
  emitter.next(1);
  log.info("emitter.complete()");
  emitter.complete();
}).subscribeOn(Schedulers.parallel());
log.info("After Flux.create()");

log.info("Before Flux.subscribe()");
flux.subscribe(
  next -> log.info("subscriber.onNext({})", next),
  error -> log.info("subscriber.onError({})", error),
  () -> log.info("subscriber.onComplete()")
);
log.info("After Flux.subscribe()");
```

```
[main] - Before Flux.create()
[main] - After Flux.create()
[main] - Before Flux.subscribe()
[main] - After Flux.subscribe()
```

Reactive Programming

SubscribeOn

```
log.info("Before Flux.create()");
Flux<Integer> flux = Flux.create(emitter -> {
  log.info("emitter.next({})", 1);
  emitter.next(1);
  log.info("emitter.complete()");
  emitter.complete();
}).subscribeOn(Schedulers.parallel());
log.info("After Flux.create()");

log.info("Before Flux.subscribe()");
flux.subscribe(
  next -> log.info("subscriber.onNext({})", next),
  error -> log.info("subscriber.onError({})", error),
  () -> log.info("subscriber.onComplete()")
);
log.info("After Flux.subscribe()");
```

```
[main] - Before Flux.create()
[main] - After Flux.create()
[main] - Before Flux.subscribe()
[main] - After Flux.subscribe()
[par1] - emitter.next(1)
```

Reactive Programming

SubscribeOn

```
log.info("Before Flux.create()");
Flux<Integer> flux = Flux.create(emitter -> {
  log.info("emitter.next({})", 1);
  emitter.next(1);
  log.info("emitter.complete()");
  emitter.complete();
}).subscribeOn(Schedulers.parallel());
log.info("After Flux.create()");

log.info("Before Flux.subscribe()");
flux.subscribe(
  next -> log.info("subscriber.onNext({})", next),
  error -> log.info("subscriber.onError({})", error),
  () -> log.info("subscriber.onComplete()")
);
log.info("After Flux.subscribe()");
```

```
[main] - Before Flux.create()
[main] - After Flux.create()
[main] - Before Flux.subscribe()
[main] - After Flux.subscribe()
[par1] - emitter.next(1)
[par1] - subscriber.onNext(1)
```

Reactive Programming

SubscribeOn

```
log.info("Before Flux.create()");
Flux<Integer> flux = Flux.create(emitter -> {
  log.info("emitter.next({})", 1);
  emitter.next(1);
  log.info("emitter.complete()");
  emitter.complete();
}).subscribeOn(Schedulers.parallel());
log.info("After Flux.create()");

log.info("Before Flux.subscribe()");
flux.subscribe(
  next -> log.info("subscriber.onNext({})", next),
  error -> log.info("subscriber.onError({})", error),
  () -> log.info("subscriber.onComplete()")
);
log.info("After Flux.subscribe()");
```

```
[main] - Before Flux.create()
[main] - After Flux.create()
[main] - Before Flux.subscribe()
[main] - After Flux.subscribe()
[par1] - emitter.next(1)
[par1] - subscriber.onNext(1)
[par1] - emitter.complete()
```

Reactive Programming

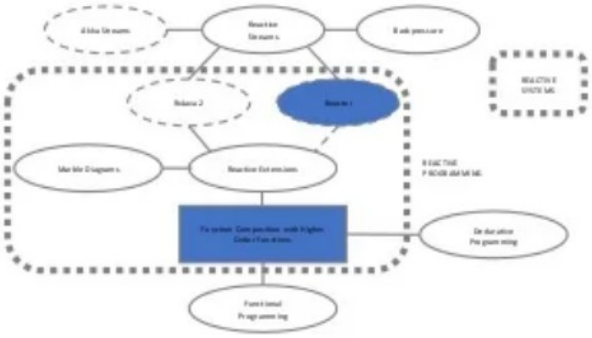
SubscribeOn

```
log.info("Before Flux.create()");
Flux<Integer> flux = Flux.create(emitter -> {
  log.info("emitter.next({})", 1);
  emitter.next(1);
  log.info("emitter.complete()");
  emitter.complete();
}).subscribeOn(Schedulers.parallel());
log.info("After Flux.create()");

log.info("Before Flux.subscribe()");
flux.subscribe(
  next -> log.info("subscriber.onNext({})", next),
  error -> log.info("subscriber.onError({})", error),
  () -> log.info("subscriber.onComplete()")
);
log.info("After Flux.subscribe()");
```

```
[main] - Before Flux.create()
[main] - After Flux.create()
[main] - Before Flux.subscribe()
[main] - After Flux.subscribe()
[par1] - emitter.next(1)
[par1] - subscriber.onNext(1)
[par1] - emitter.complete()
[par1] - subscriber.onComplete()
```

Reactive Programming

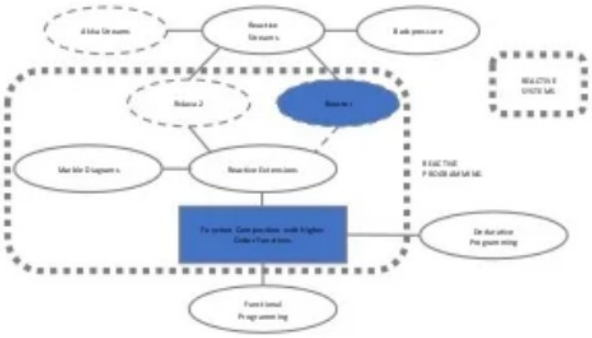
SubscribeOn



```
Log.info("Before Flux.fromIterable()");
Flux<User> users = Flux
  .fromIterable(cwids)
  .flatMap(cwid -> Mono
    .fromCallable(() -> loadUser(cwid)))
                                        ;
```

Reactive Programming

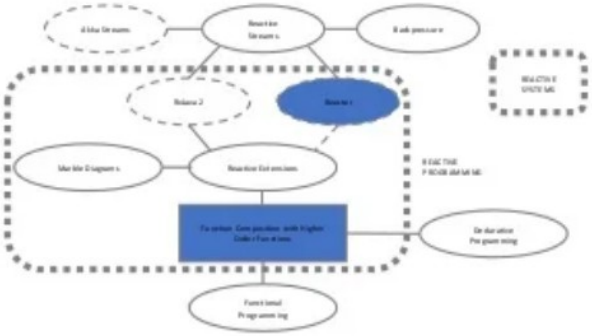
SubscribeOn



```
log.info("Before Flux.fromIterable()");
Flux<User> users = Flux
  .fromIterable(cwids)
  .flatMap(cwid -> Mono
    .fromCallable(() -> loadUser(cwid)))
  .subscribeOn(Schedulers.parallel());
```

Reactive Programming

SubscribeOn

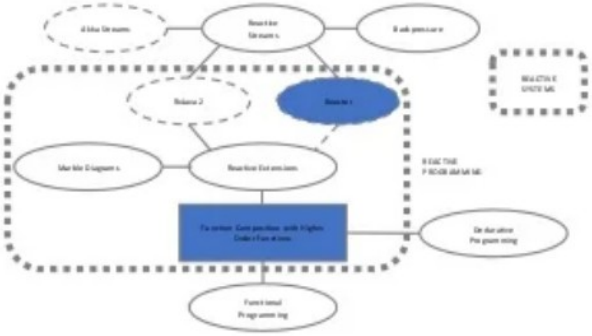


```
log.info("Before Flux.fromIterable()");
Flux<User> users = Flux
  .fromIterable(cwids)
  .flatMap(cwid -> Mono
    .fromCallable(() -> loadUser(cwid)))
  .subscribeOn(Schedulers.parallel());
```

```
[main] - Before Flux.fromIterable()
```

Reactive Programming

SubscribeOn

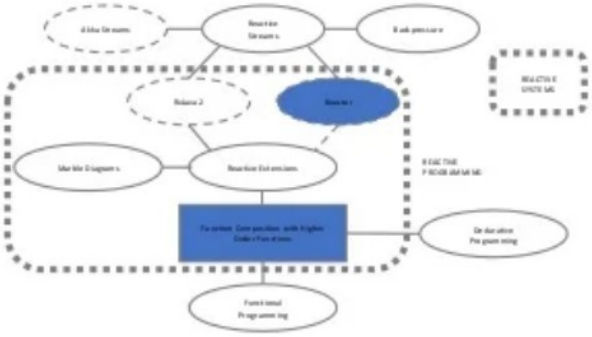


```
log.info("Before Flux.fromIterable()");
Flux<User> users = Flux
  .fromIterable(cwids)
  .flatMap(cwid -> Mono
    .fromCallable(() -> loadUser(cwid)))
  .subscribeOn(Schedulers.parallel());
```

```
[main] - Before Flux.fromIterable()
[par1] - loadUser(7896c6eb)
```

Reactive Programming

SubscribeOn

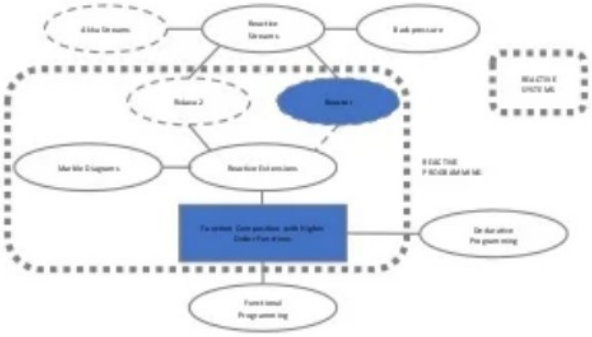


```
log.info("Before Flux.fromIterable()");
Flux<User> users = Flux
  .fromIterable(cwids)
  .flatMap(cwid -> Mono
    .fromCallable(() -> loadUser(cwid)))
  .subscribeOn(Schedulers.parallel());
```

```
[main] - Before Flux.fromIterable()
[par1] - loadUser(7896c6eb)
[par1] - loadUser(75ad9445)
```

Reactive Programming

SubscribeOn

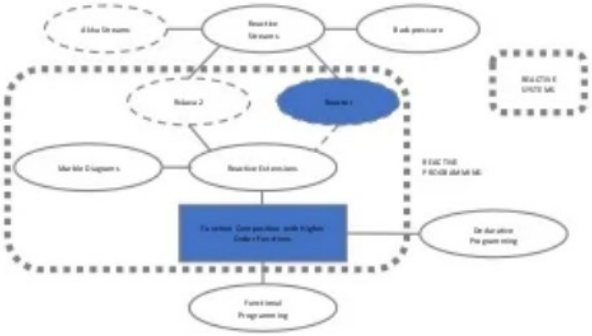


```
log.info("Before Flux.fromIterable()");
Flux<User> users = Flux
  .fromIterable(cwids)
  .flatMap(cwid -> Mono
    .fromCallable(() -> loadUser(cwid)))
  .subscribeOn(Schedulers.parallel());
```

```
[main] - Before Flux.fromIterable()
[par1] - loadUser(7896c6eb)
[par1] - loadUser(75ad9445)
[par1] - loadUser(e2d4c51d)
```

Reactive Programming

SubscribeOn

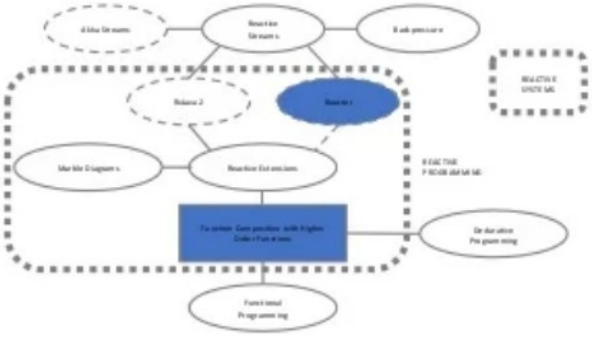


```
log.info("Before Flux.fromIterable()");
Flux<User> users = Flux
  .fromIterable(cwids)
  .flatMap(cwid -> Mono
    .fromCallable(() -> loadUser(cwid)))
  .subscribeOn(Schedulers.parallel());

StepVerifier.create(users)
```

```
[main] - Before Flux.fromIterable()
[par1] - loadUser(7896c6eb)
[par1] - loadUser(75ad9445)
[par1] - loadUser(e2d4c51d)
```

Reactive Programming

SubscribeOn

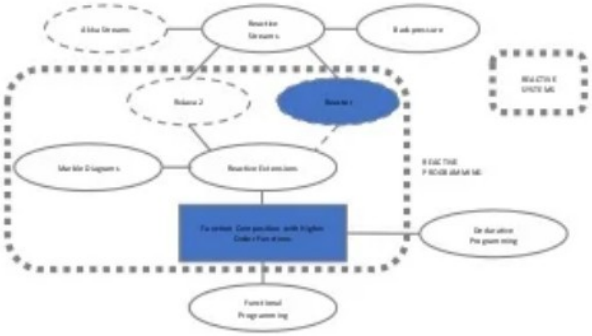


```
log.info("Before Flux.fromIterable()");
Flux<User> users = Flux
  .fromIterable(cwids)
  .flatMap(cwid -> Mono
    .fromCallable(() -> loadUser(cwid)))
  .subscribeOn(Schedulers.parallel());

StepVerifier.create(users)
  .expectNextMatches(hasCwid(cwids.get(0)))
```

```
[main] - Before Flux.fromIterable()
[par1] - loadUser(7896c6eb)
[par1] - loadUser(75ad9445)
[par1] - loadUser(e2d4c51d)
```

Reactive Programming

SubscribeOn

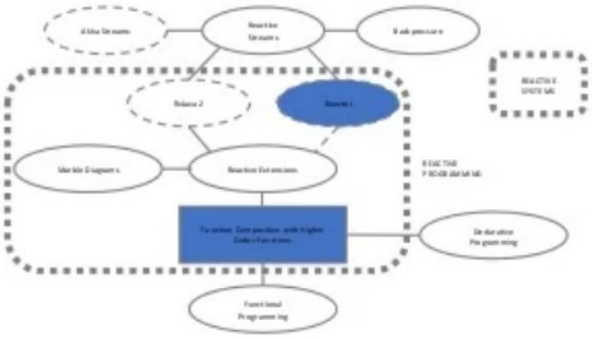


```
log.info("Before Flux.fromIterable()");
Flux<User> users = Flux
  .fromIterable(cwids)
  .flatMap(cwid -> Mono
    .fromCallable(() -> loadUser(cwid)))
  .subscribeOn(Schedulers.parallel());

StepVerifier.create(users)
  .expectNextMatches(hasCwid(cwids.get(0)))
  .expectNextMatches(hasCwid(cwids.get(1)))
```

```
[main] - Before Flux.fromIterable()
[par1] - loadUser(7896c6eb)
[par1] - loadUser(75ad9445)
[par1] - loadUser(e2d4c51d)
```

Reactive Programming

SubscribeOn

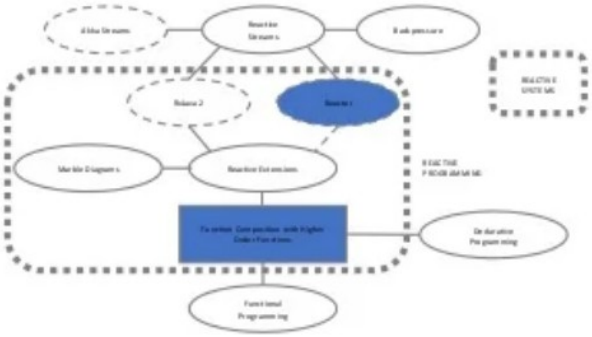


```
log.info("Before Flux.fromIterable()");
Flux<User> users = Flux
  .fromIterable(cwids)
  .flatMap(cwid -> Mono
    .fromCallable(() -> loadUser(cwid)))
  .subscribeOn(Schedulers.parallel());

StepVerifier.create(users)
  .expectNextMatches(hasCwid(cwids.get(0)))
  .expectNextMatches(hasCwid(cwids.get(1)))
  .expectNextMatches(hasCwid(cwids.get(2)))
```

```
[main] - Before Flux.fromIterable()
[par1] - loadUser(7896c6eb)
[par1] - loadUser(75ad9445)
[par1] - loadUser(e2d4c51d)
```

Reactive Programming

SubscribeOn

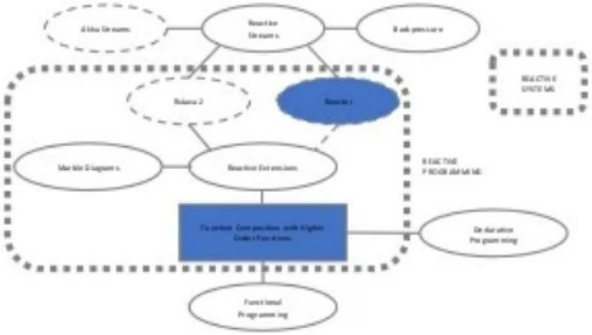


```
log.info("Before Flux.fromIterable()");
Flux<User> users = Flux
  .fromIterable(cwids)
  .flatMap(cwid -> Mono
    .fromCallable(() -> loadUser(cwid)))
  .subscribeOn(Schedulers.parallel());

StepVerifier.create(users)
  .expectNextMatches(hasCwid(cwids.get(0)))
  .expectNextMatches(hasCwid(cwids.get(1)))
  .expectNextMatches(hasCwid(cwids.get(2)))
  ...
  .verifyComplete();
```

```
[main] - Before Flux.fromIterable()
[par1] - loadUser(7896c6eb)
[par1] - loadUser(75ad9445)
[par1] - loadUser(e2d4c51d)
```

Reactive Programming

SubscribeOn

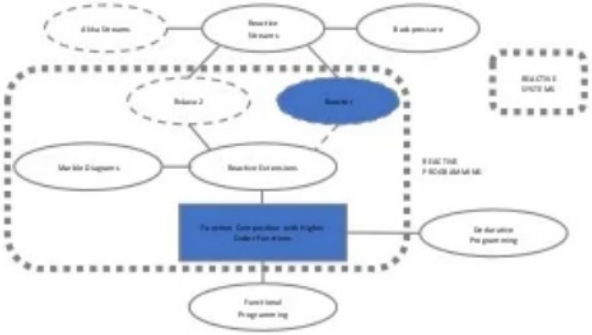


```
Log.info("Before Flux.fromIterable()");
Flux<User> users = Flux
  .fromIterable(cwids)
  .flatMap(cwid -> Mono
    .fromCallable(() -> loadUser(cwid)))
  .subscribeOn(Schedulers.parallel());

StepVerifier.create(users)
  .expectNextMatches(hasCwid(cwids.get(0)))
  .expectNextMatches(hasCwid(cwids.get(1)))
  .expectNextMatches(hasCwid(cwids.get(2)))
  ...
  .verifyComplete();
```

```
[main] - Before Flux.fromIterable()
[par1] - loadUser(7896c6eb)
[par1] - loadUser(75ad9445)
[par1] - loadUser(e2d4c51d)
```

Reactive Programming

SubscribeOn

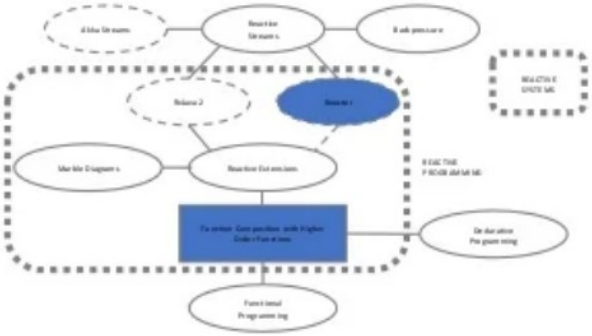


```
log.info("Before Flux.fromIterable()");
Flux<User> users = Flux
  .fromIterable(cwids)
  .flatMap(cwid -> Mono
    .fromCallable(() -> loadUser(cwid)))
  .subscribeOn(Schedulers.parallel());

StepVerifier.create(users)
  .expectNextMatches(hasCwid(cwids.get(0)))
  .expectNextMatches(hasCwid(cwids.get(1)))
  .expectNextMatches(hasCwid(cwids.get(2)))
  ...
  .verifyComplete();
```

```
[main] - Before Flux.fromIterable()
[par1] - loadUser(7896c6eb)
[par1] - loadUser(75ad9445)
[par1] - loadUser(e2d4c51d)
```

Reactive Programming

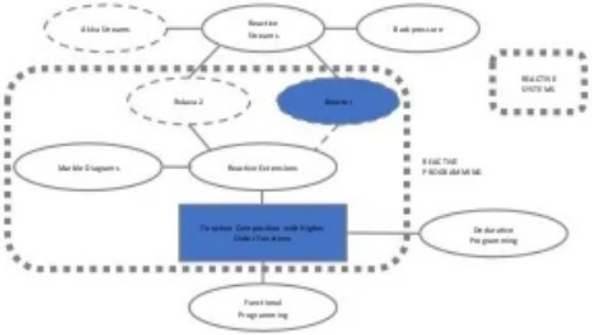
SubscribeOn



```
Log.info("Before Flux.fromIterable()");
Flux<User> users = Flux
  .fromIterable(cwids)
  .flatMap(cwid -> Mono
    .fromCallable(() -> loadUser(cwid))
                                        );
```

Reactive Programming

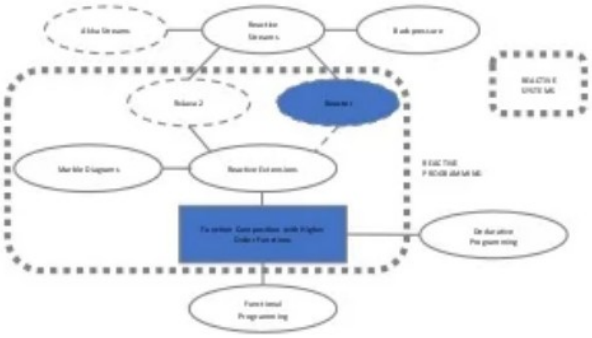
SubscribeOn



```
log.info("Before Flux.fromIterable()");
Flux<User> users = Flux
  .fromIterable(cwids)
  .flatMap(cwid -> Mono
    .fromCallable(() -> loadUser(cwid))
    .subscribeOn(Schedulers.parallel()));
```

Reactive Programming

SubscribeOn

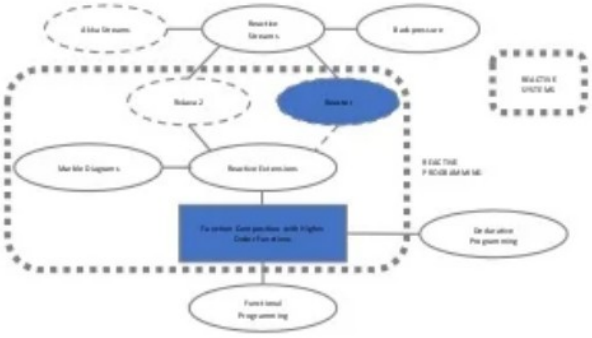


```
log.info("Before Flux.fromIterable()");
Flux<User> users = Flux
  .fromIterable(cwids)
  .flatMap(cwid -> Mono
    .fromCallable(() -> loadUser(cwid))
    .subscribeOn(Schedulers.parallel()));
```

```
[main] - Before Flux.fromIterable()
```

Reactive Programming

SubscribeOn

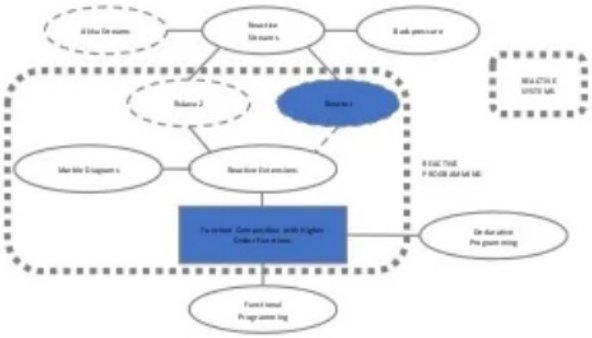


```
log.info("Before Flux.fromIterable()");
Flux<User> users = Flux
  .fromIterable(cwids)
  .flatMap(cwid -> Mono
    .fromCallable(() -> loadUser(cwid))
    .subscribeOn(Schedulers.parallel()));
```

```
[main] - Before Flux.fromIterable()
[par1] - loadUser(7896c6eb)
```

Reactive Programming

SubscribeOn

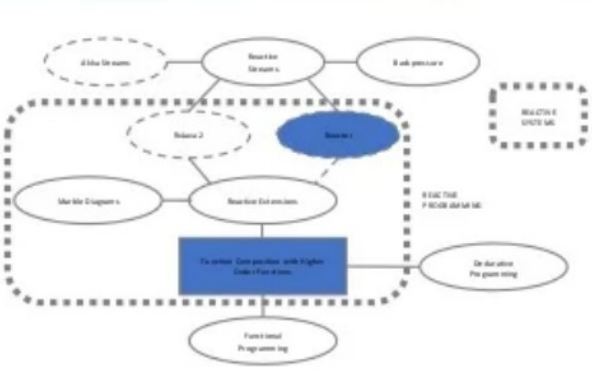


```
log.info("Before Flux.fromIterable()");
Flux<User> users = Flux
  .fromIterable(cwids)
  .flatMap(cwid -> Mono
    .fromCallable(() -> loadUser(cwid))
    .subscribeOn(Schedulers.parallel()));
```

```
[main] - Before Flux.fromIterable()
[par1] - loadUser(7896c6eb)
[par2] - loadUser(75ad9445)
```

Reactive Programming

SubscribeOn

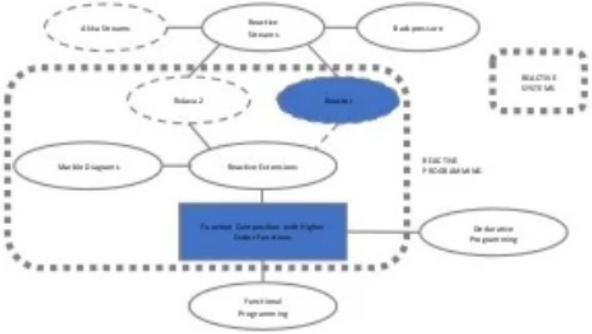


```
log.info("Before Flux.fromIterable()");
Flux<User> users = Flux
  .fromIterable(cwids)
  .flatMap(cwid -> Mono
    .fromCallable(() -> loadUser(cwid))
    .subscribeOn(Schedulers.parallel()));
```

```
[main] - Before Flux.fromIterable()
[par1] - loadUser(7896c6eb)
[par2] - loadUser(75ad9445)
[par3] - loadUser(e2d4c51d)
```

Reactive Programming

SubscribeOn

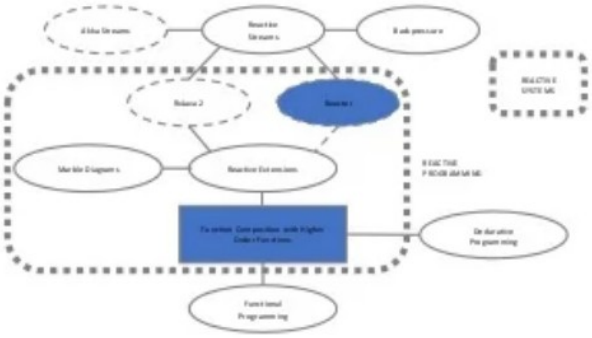


```
log.info("Before Flux.fromIterable()");
Flux<User> users = Flux
  .fromIterable(cwids)
  .flatMap(cwid -> Mono
    .fromCallable(() -> loadUser(cwid))
    .subscribeOn(Schedulers.parallel()));

StepVerifier.create(users)
  .expectNextMatches(hasCwid(cwids.get(0)))
  .expectNextMatches(hasCwid(cwids.get(1)))
  .expectNextMatches(hasCwid(cwids.get(2)))
  ...
  .verifyComplete();
```

```
[main] - Before Flux.fromIterable()
[par1] - loadUser(7896c6eb)
[par2] - loadUser(75ad9445)
[par3] - loadUser(e2d4c51d)
```

Reactive Programming

SubscribeOn

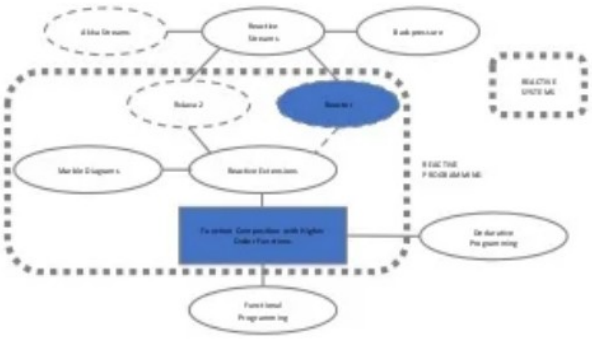


```
Log.info("Before Flux.fromIterable()");
Flux<User> users = Flux
  .fromIterable(cwids)
  .flatMap(cwid -> Mono
    .fromCallable(() -> loadUser(cwid))
    .subscribeOn(Schedulers.parallel()));

StepVerifier.create(users)
  .expectNextMatches(hasCwid(cwids.get(0)))
  .expectNextMatches(hasCwid(cwids.get(1)))
  .expectNextMatches(hasCwid(cwids.get(2)))
  ...
  .verifyComplete();
```

```
[main] - Before Flux.fromIterable()
[par1] - loadUser(7896c6eb)
[par2] - loadUser(75ad9445)
[par3] - loadUser(e2d4c51d)
```

Reactive Programming

SubscribeOn

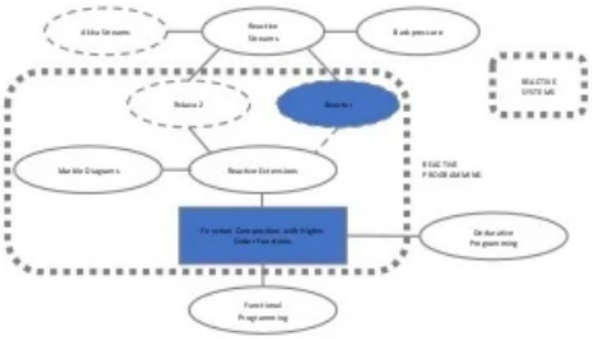


```
log.info("Before Flux.fromIterable()");
Flux<User> users = Flux
  .fromIterable(cwids)
  .flatMap(cwid -> Mono
    .fromCallable(() -> loadUser(cwid))
    .subscribeOn(Schedulers.parallel()));

StepVerifier.create(users)
  .expectNextMatches(hasCwid(cwids.get(0)))
  .expectNextMatches(hasCwid(cwids.get(1)))
  .expectNextMatches(hasCwid(cwids.get(2)))
  ...
  .verifyComplete();
```

```
[main] - Before Flux.fromIterable()
[par1] - loadUser(7896c6eb)
[par2] - loadUser(75ad9445)
[par3] - loadUser(e2d4c51d)
```

Reactive Programming

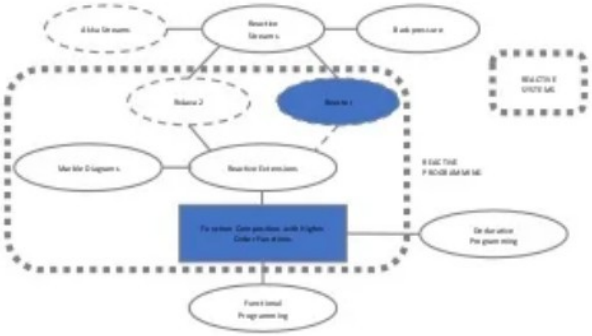
Virtual Time



```
log.info("Before Flux.fromIterable()");
          Flux<User>  users =       Flux
  .fromIterable(cwids)
  .flatMap(cwid -> Mono
    .fromCallable(() -> loadUser(cwid))
    .subscribeOn(Schedulers.parallel()));

StepVerifier
```

Reactive Programming

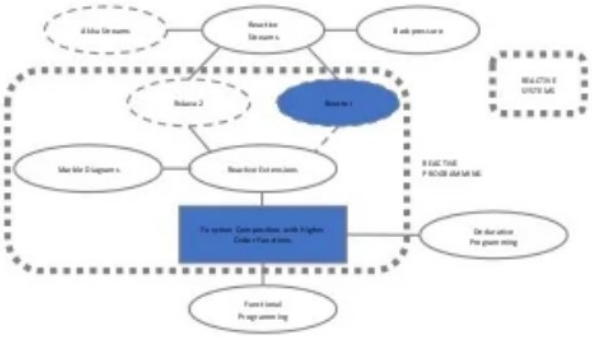
Virtual Time



```
log.info("Before Flux.fromIterable()");
          Flux<User>  users =       Flux
  .fromIterable(cwids)
  .flatMap(cwid -> Mono
    .fromCallable(() -> loadUser(cwid))
    .subscribeOn(Schedulers.parallel()));

StepVerifier.withVirtualTime(users)
```

Reactive Programming

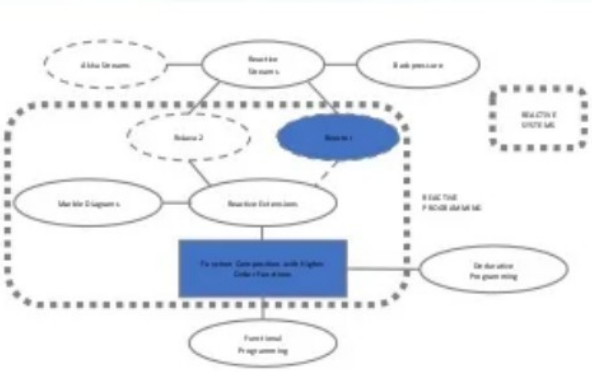
Virtual Time



```
Log.info("Before Flux.fromIterable()");
          Flux<User>  users =       Flux
  .fromIterable(cwids)
  .flatMap(cwid -> Mono
    .fromCallable(() -> loadUser(cwid))
    .subscribeOn(Schedulers.parallel()));

StepVerifier.withVirtualTime(users)
  .thenAwait()
```

Reactive Programming

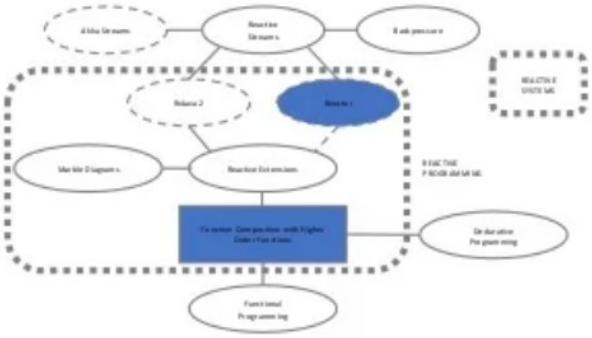
Virtual Time



```
Log.info("Before Flux.fromIterable()");
          Flux<User>  users =       Flux
  .fromIterable(cwids)
  .flatMap(cwid -> Mono
    .fromCallable(() -> loadUser(cwid))
    .subscribeOn(Schedulers.parallel()));

StepVerifier.withVirtualTime(users)
  .thenAwait()
  .expectNextMatches(hasCwid(cwids.get(0)))
```

Reactive Programming

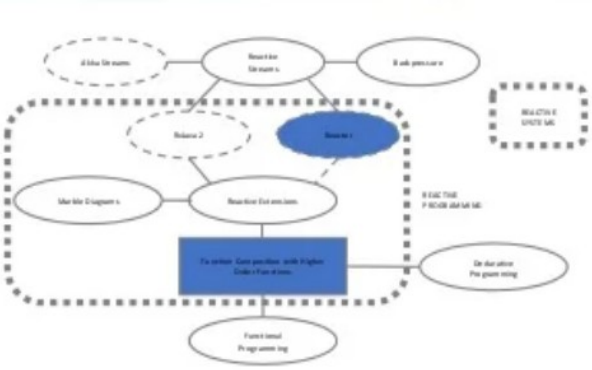
Virtual Time



```
Log.info("Before Flux.fromIterable()");
          Flux<User>  users =       Flux
  .fromIterable(cwids)
  .flatMap(cwid -> Mono
    .fromCallable(() -> loadUser(cwid))
    .subscribeOn(Schedulers.parallel()));

StepVerifier.withVirtualTime(users)
  .thenAwait()
  .expectNextMatches(hasCwid(cwids.get(0)))
  .expectNextMatches(hasCwid(cwids.get(1)))
```

Reactive Programming

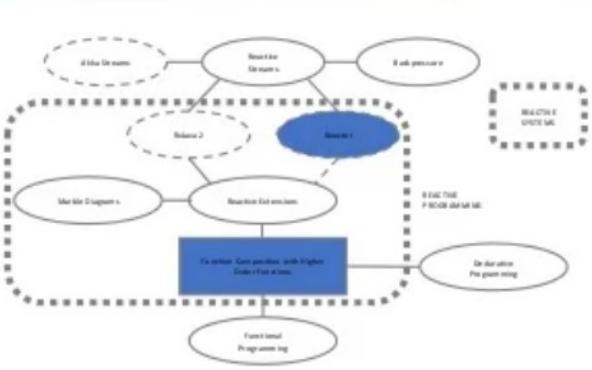
Virtual Time



```
log.info("Before Flux.fromIterable()");
          Flux<User>  users =       Flux
  .fromIterable(cwids)
  .flatMap(cwid -> Mono
    .fromCallable(() -> loadUser(cwid))
    .subscribeOn(Schedulers.parallel()));

StepVerifier.withVirtualTime(users)
  .thenAwait()
  .expectNextMatches(hasCwid(cwids.get(0)))
  .expectNextMatches(hasCwid(cwids.get(1)))
  .expectNextMatches(hasCwid(cwids.get(2)))
```

Reactive Programming

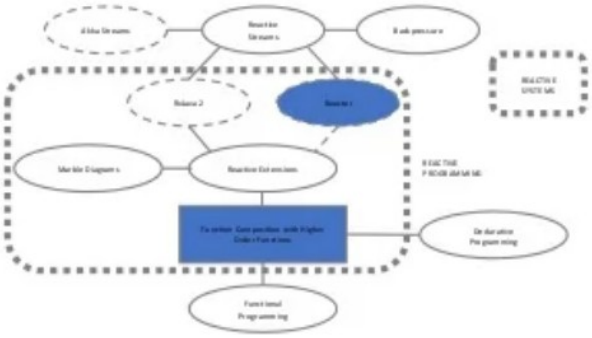
Virtual Time



```
log.info("Before Flux.fromIterable()");
          Flux<User>  users =       Flux
  .fromIterable(cwids)
  .flatMap(cwid -> Mono
    .fromCallable(() -> loadUser(cwid))
    .subscribeOn(Schedulers.parallel()));

StepVerifier.withVirtualTime(users)
  .thenAwait()
  .expectNextMatches(hasCwid(cwids.get(0)))
  .expectNextMatches(hasCwid(cwids.get(1)))
  .expectNextMatches(hasCwid(cwids.get(2)))
  ...
  .verifyComplete();
```

Reactive Programming

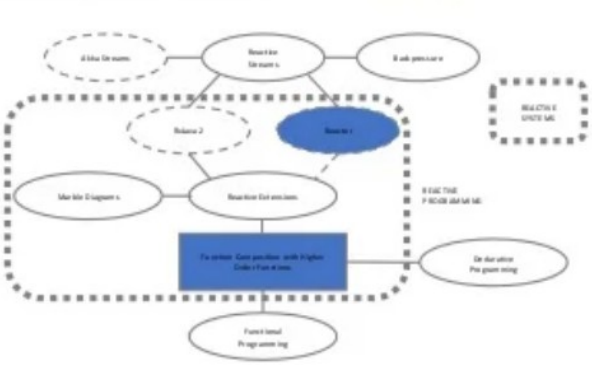
Virtual Time



```
log.info("Before Flux.fromIterable()");
Supplier<Flux<User>> users = () -> Flux
  .fromIterable(cwids)
  .flatMap(cwid -> Mono
    .fromCallable(() -> loadUser(cwid))
    .subscribeOn(Schedulers.parallel()));

StepVerifier.withVirtualTime(users)
  .thenAwait()
  .expectNextMatches(hasCwid(cwids.get(0)))
  .expectNextMatches(hasCwid(cwids.get(1)))
  .expectNextMatches(hasCwid(cwids.get(2)))
  ...
  .verifyComplete();
```

Reactive Programming

Virtual Time



```
log.info("Before Flux.fromIterable()");
Supplier<Flux<User>> users = () -> Flux
  .fromIterable(cwids)
  .flatMap(cwid -> Mono
    .fromCallable(() -> loadUser(cwid))
    .subscribeOn(Schedulers.parallel()));

StepVerifier.withVirtualTime(users)
  .thenAwait()
  .expectNextMatches(hasCwid(cwids.get(0)))
  .expectNextMatches(hasCwid(cwids.get(1)))
  .expectNextMatches(hasCwid(cwids.get(2)))
  ...
  .verifyComplete();
```

Reactive Programming

Let‘s go wild ...

```
                                        Flux
.fromIterable(cwids)
.flatMap(cwid -> Mono.fromCallable(() -> loadUser(cwid))









  .subscribeOn(Schedulers.parallel()));
```

Reactive Programming

Let‘s go wild ...

```
                                        Flux
.fromIterable(cwids)
.flatMap(cwid -> Mono.fromCallable(() -> loadUser(cwid))
```

```
  .subscribeOn(Schedulers.parallel()));
```

CONCURRENCY

Reactive Programming

Let‘s go wild ...

TIMEOUT

```
                                        Flux
.fromIterable(cwids)
.flatMap(cwid -> Mono.fromCallable(() -> loadUser(cwid))
  .timeout(Duration.ofMillis(250), Schedulers.single())









  .subscribeOn(Schedulers.parallel()));
```

CONCURRENCY

Reactive Programming

Let‘s go wild ...

RETRY

TIMEOUT

```
Flux
.fromIterable(cwids)
.flatMap(cwid -> Mono.fromCallable(() -> loadUser(cwid))
  .timeout(Duration.ofMillis(250), Schedulers.single())
  .retry(numRetries)

  .subscribeOn(Schedulers.parallel()));
```

CONCURRENCY

Reactive Programming

Let‘s go wild ...

RETRY WITH EXPONENTIAL BACKOFF

TIMEOUT

```
                                        Flux
  .fromIterable(cwids)
  .flatMap(cwid -> Mono.fromCallable(() -> loadUser(cwid))
    .timeout(Duration.ofMillis(250), Schedulers.single())
    .retryWhen(errors -> errors.zipWith(Flux.range(0, MAX_VALUE), (err, idx) -> {
      if (idx < numRetries) {
        return Mono.just(err).delayElement(withExponentialBackoff(idx));
      } else {
        return Mono.error(err);
      }
    }).flatMap(Function.identity()))


    .subscribeOn(Schedulers.parallel()));
```

CONCURRENCY

Reactive Programming

Let‘s go wild ...

RETRY WITH EXPONENTIAL BACKOFF

TIMEOUT

```
Flux
.fromIterable(cwids)
.flatMap(cwid -> Mono.fromCallable(() -> loadUser(cwid))
  .timeout(Duration.ofMillis(250), Schedulers.single())
  .retryWhen(errors -> errors.zipWith(Flux.range(0, MAX_VALUE), (err, idx) -> {
    if (idx < numRetries) {
      return Mono.just(err).delayElement(withExponentialBackoff(idx));
    } else {
      return Mono.error(err);
    }
  }).flatMap(Function.identity()))


  .subscribeOn(Schedulers.parallel()));
```

```
long ms = (long) Math.pow(10, idx);
return Duration.ofMillis(ms);
```

CONCURRENCY

Reactive Programming

Let‘s go wild ...

RETRY WITH EXPONENTIAL BACKOFF

TIMEOUT

```
                                        Flux
  .fromIterable(cwids)
  .flatMap(cwid -> Mono.fromCallable(() -> loadUser(cwid))
    .timeout(Duration.ofMillis(250), Schedulers.single())
    .retryWhen(errors -> errors.zipWith(Flux.range(0, MAX_VALUE), (err, idx) -> {
      if (idx < numRetries) {
        return Mono.just(err).delayElement(withExponentialBackoff(idx));
      } else {
        return Mono.error(err);
      }
    }).flatMap(Function.identity()))


    .subscribeOn(Schedulers.parallel()));
```

CONCURRENCY

Reactive Programming

Let‘s go wild ...

RETRY WITH EXPONENTIAL BACKOFF

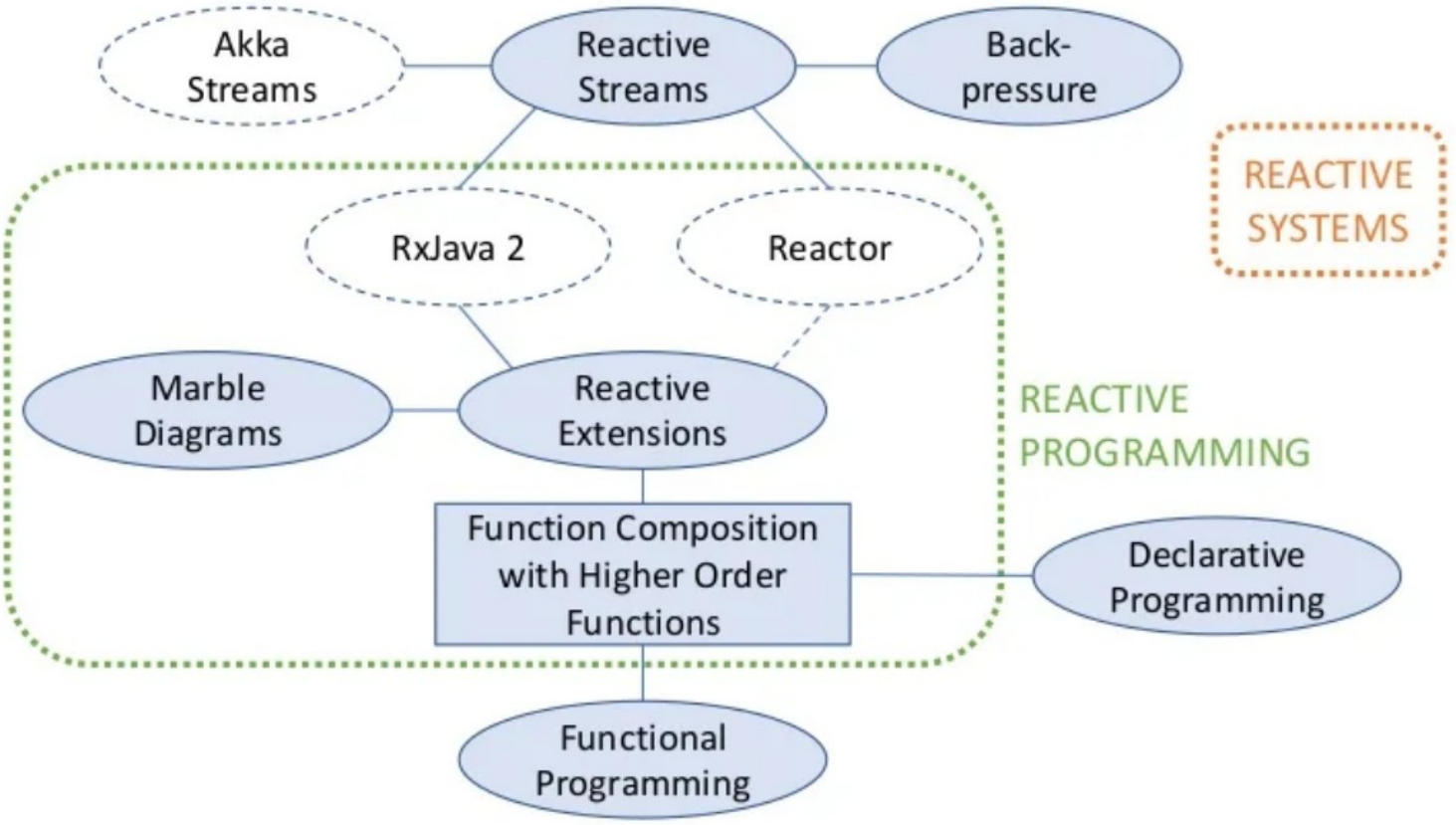
TIMEOUT

```
Flux<Either<String, User>> users = Flux
  .fromIterable(cwids)
  .flatMap(cwid -> Mono.fromCallable(() -> loadUser(cwid))
    .timeout(Duration.ofMillis(250), Schedulers.single())
    .retryWhen(errors -> errors.zipWith(Flux.range(0, MAX_VALUE), (err, idx) -> {
      if (idx < numRetries) {
        return Mono.just(err).delayElement(withExponentialBackoff(idx));
      } else {
        return Mono.error(err);
      }
    }).flatMap(Function.identity()))
    .map(Either::<String, User>right)
    .otherwiseReturn(left(cwid))
    .subscribeOn(Schedulers.parallel()));
```

CONCURRENCY

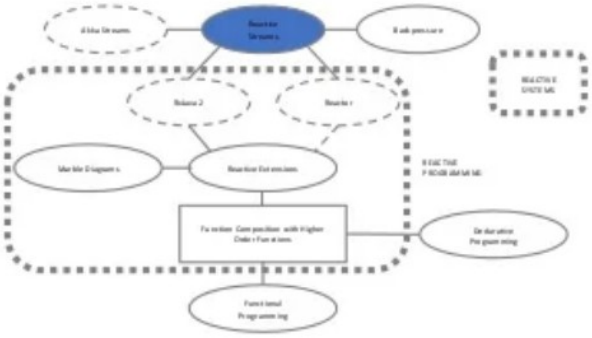
ERROR HANDLING

Reactive Programming



Reactive Programming

Reactive Streams

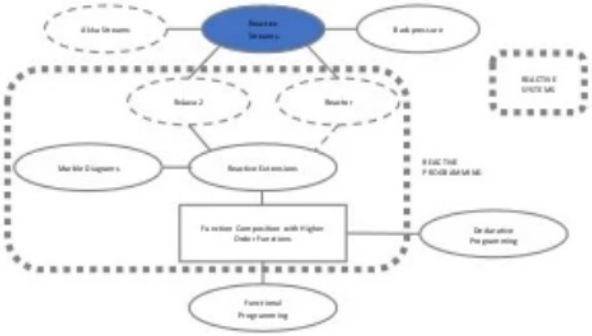


- Flow API in Java 9: `java.util.concurrent.Flow`
- Standard for asynchronous stream processing
- Pivotal, Lightbend, Netflix, Oracle, Red Hat and others

Reactive Programming

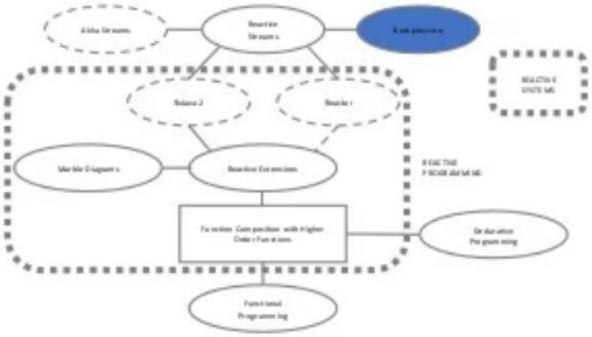
Reactive Streams

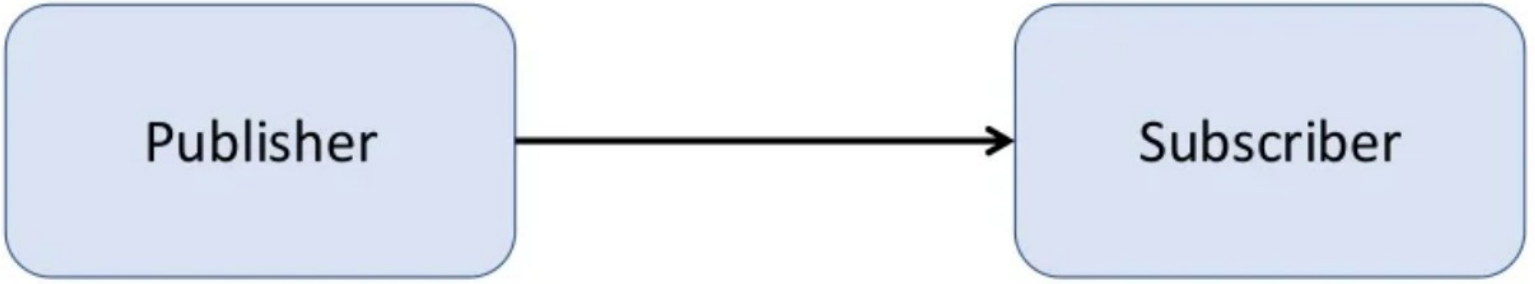
```
public class Flow {

    interface Processor<T, R> { ... }

    interface Publisher<T> { ... }

    interface Subscriber<T> { ... }

    interface Subscription { ... }

}
```



Reactive Programming

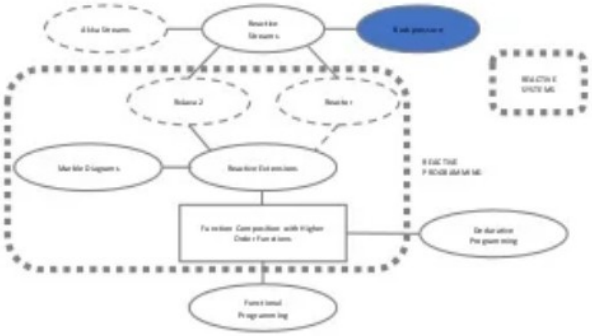
Pushing

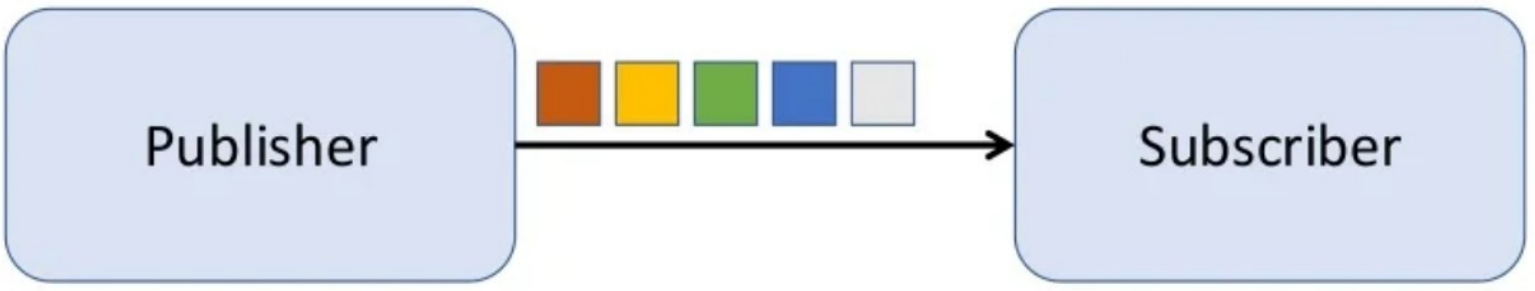




Reactive Programming

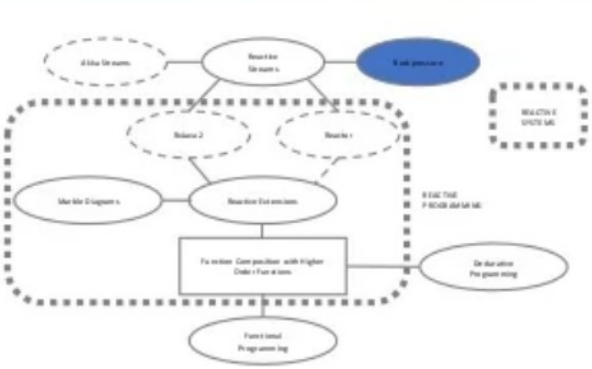
Pushing

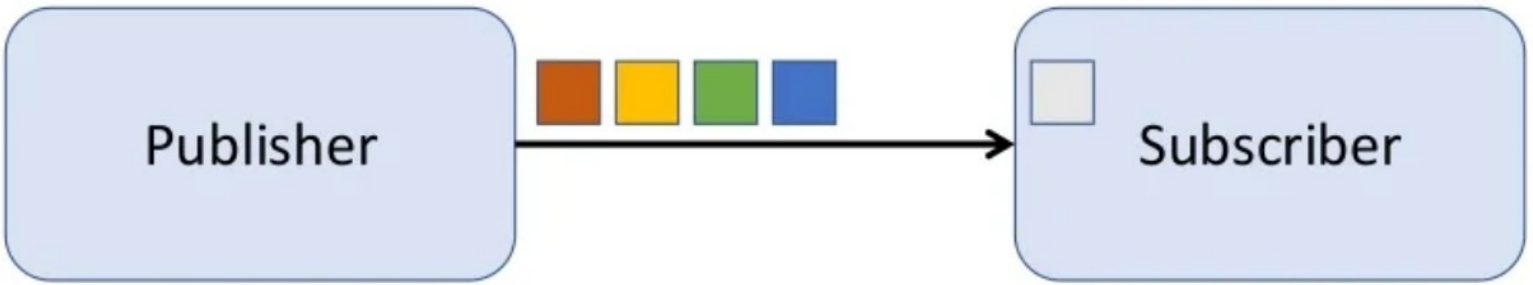




Reactive Programming

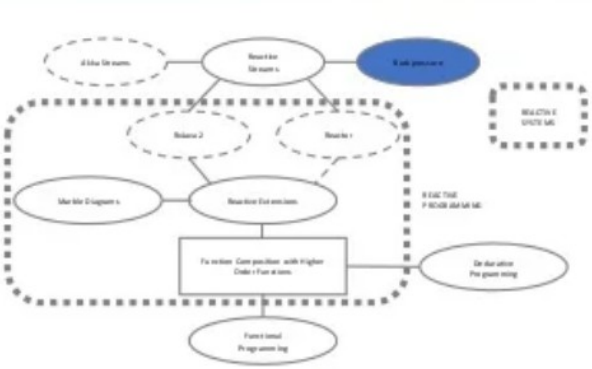
Pushing

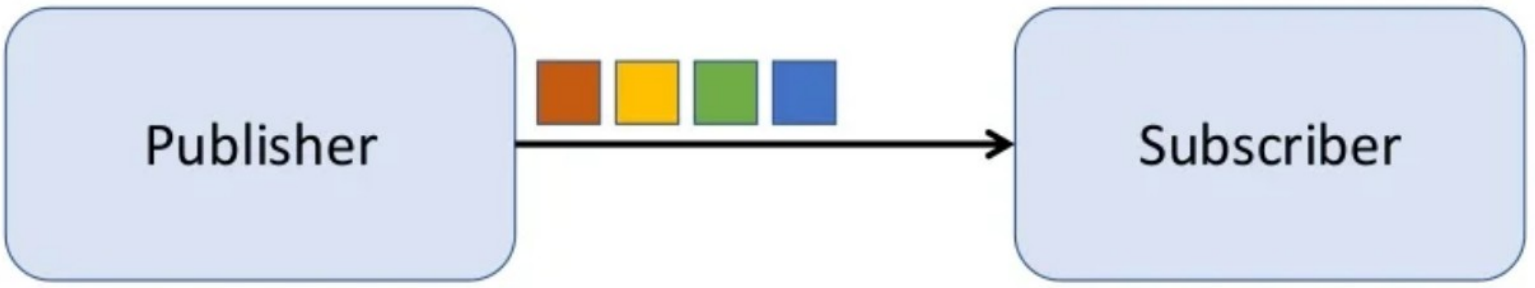




Reactive Programming

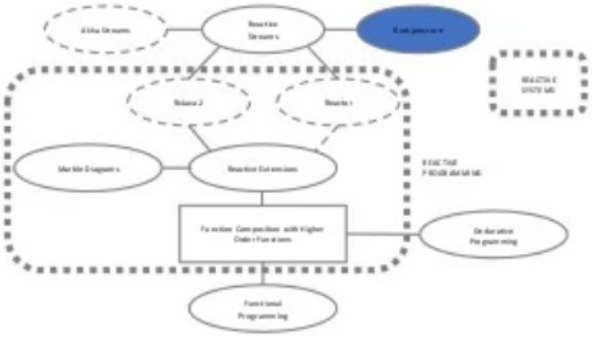
Pushing





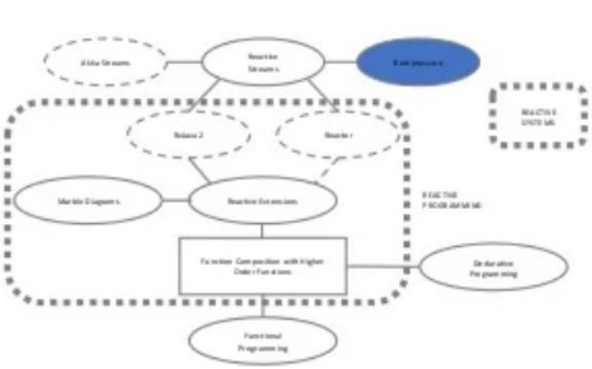
Reactive Programming

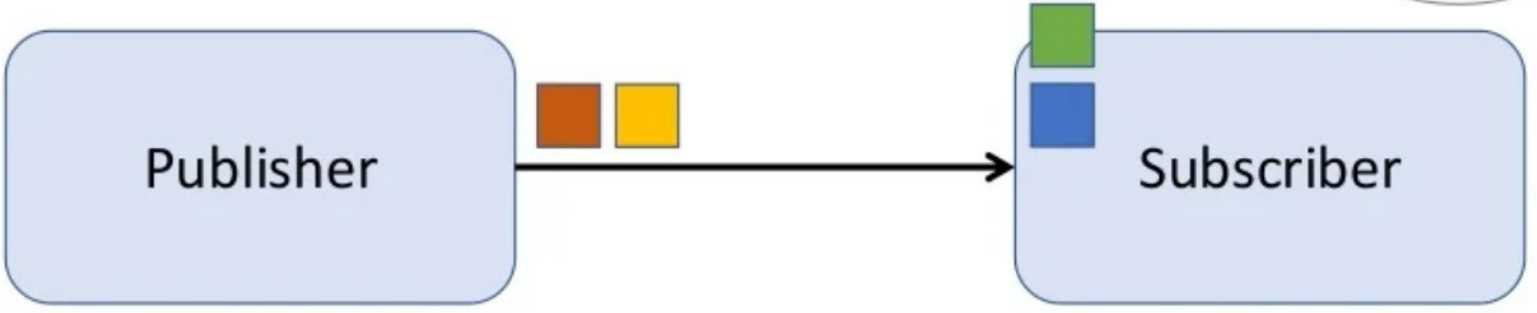
Pushing



Reactive Programming

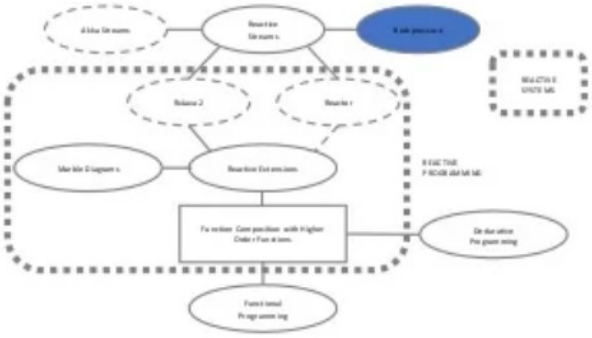
Pushing





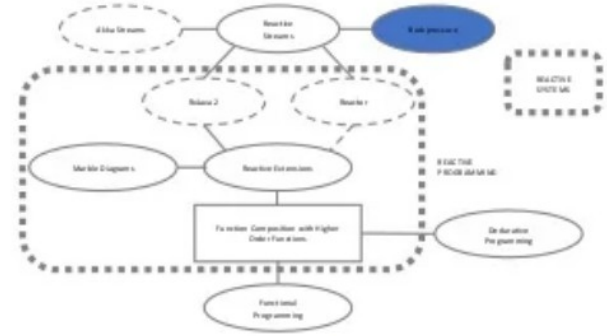
Reactive Programming

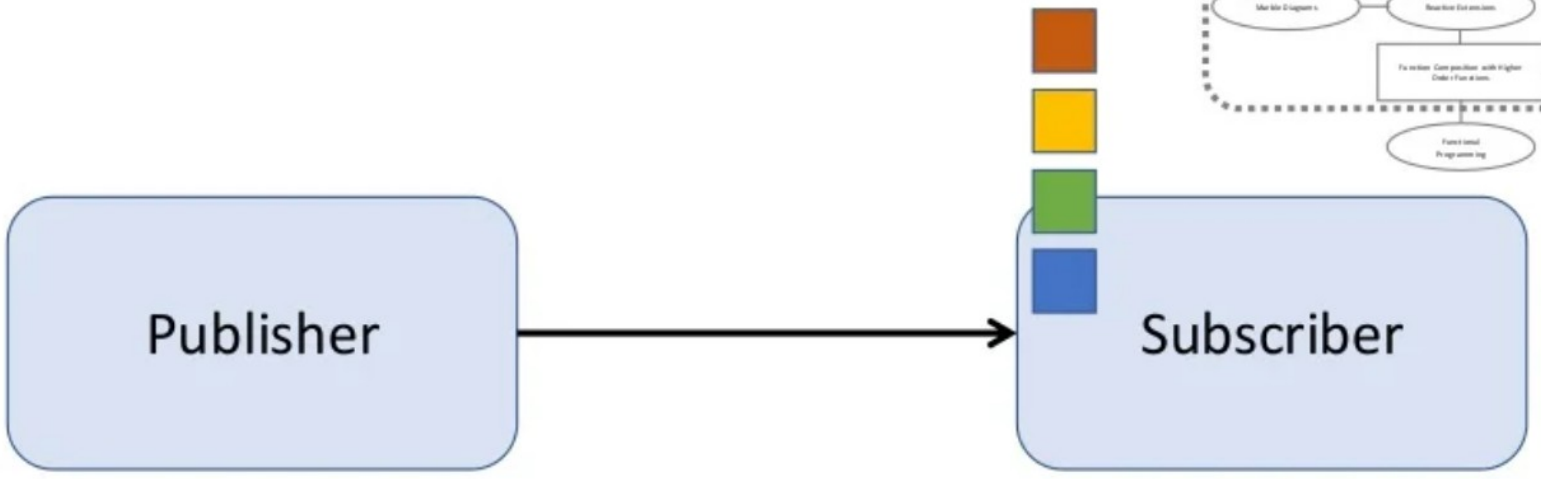
Pushing



Reactive Programming

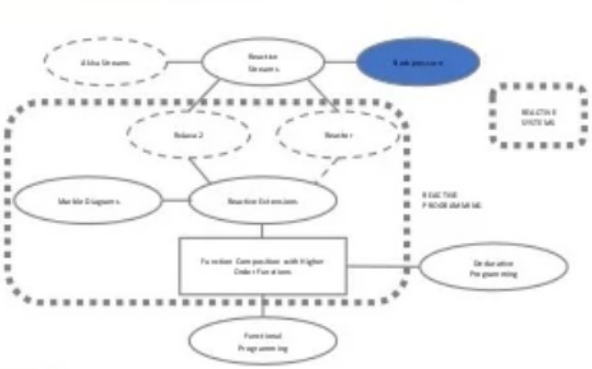
Pushing





Reactive Programming

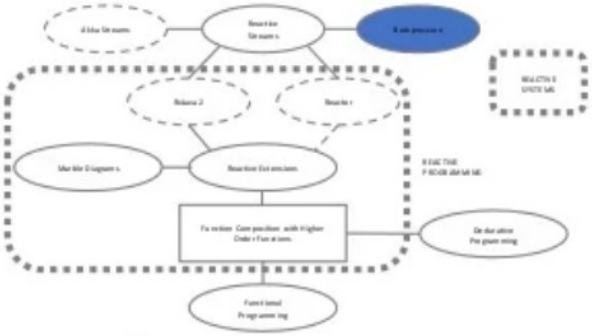
Pushing



```
ExecutorService executorService = new ThreadPoolExecutor(
  10,
  10,
  0L, TimeUnit.MILLISECONDS,
  new LinkedBlockingQueue<>(500)
);
```

Reactive Programming

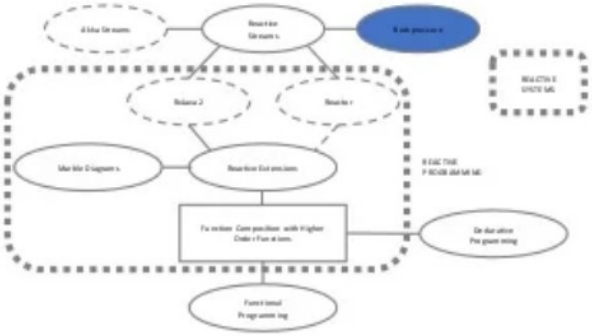
Pushing



```
List<Future<User>> eventualUsers = cwids.stream()
  .map(cwid -> executorService.submit(() -> loadUser(cwid)))
  .collect(toList());

List<User> users = eventualUsers.stream()
  .map(eventualUser -> {
    try { return eventualUser.get(); }
    catch (Exception e) { throw new RuntimeException(e); }
  })
  .collect(toList());
```

Reactive Programming

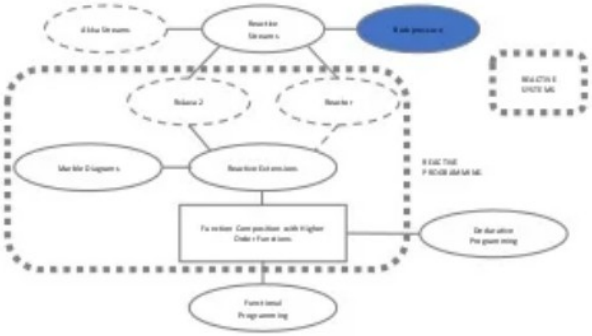
Pushing

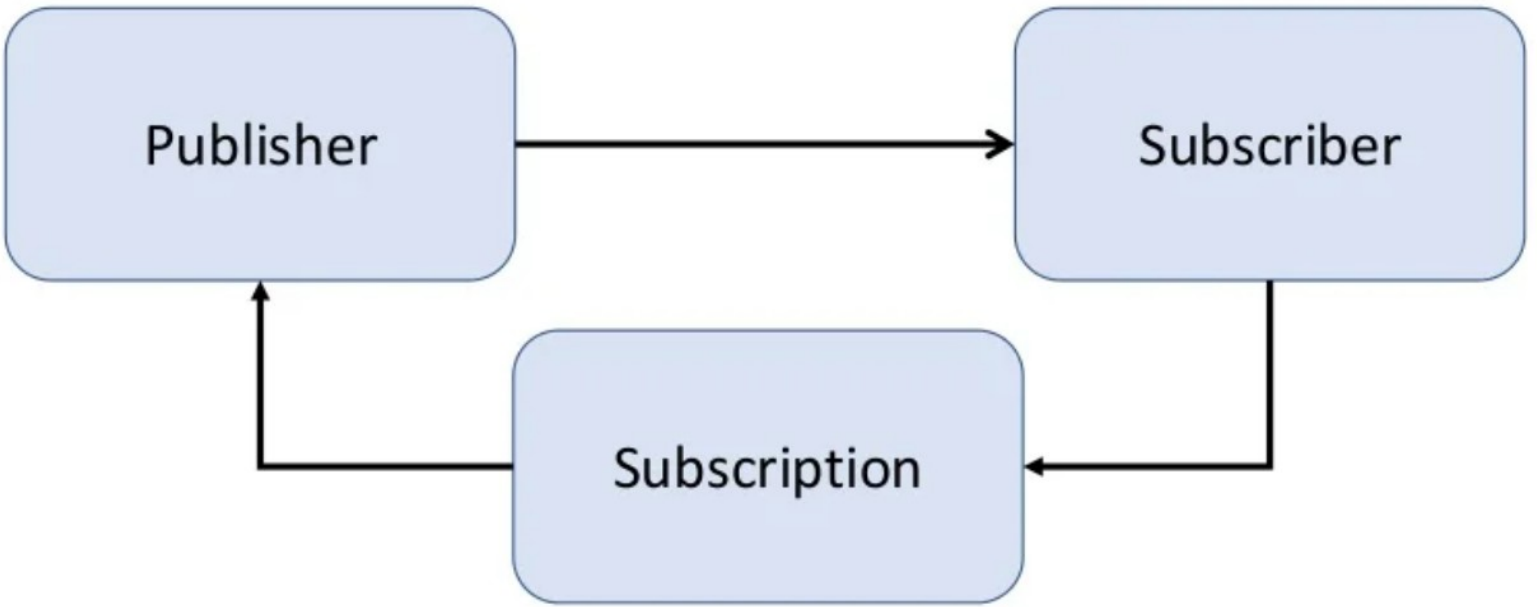


```
java.util.concurrent.RejectedExecutionException:
  Task java.util.concurrent.FutureTask@632ceb35 rejected
  from java.util.concurrent.ThreadPoolExecutor@1c93f6e1[
    Running, pool size = 10, active threads = 10,
    queued tasks = 500, completed tasks = 0
  ]
```

Reactive Programming

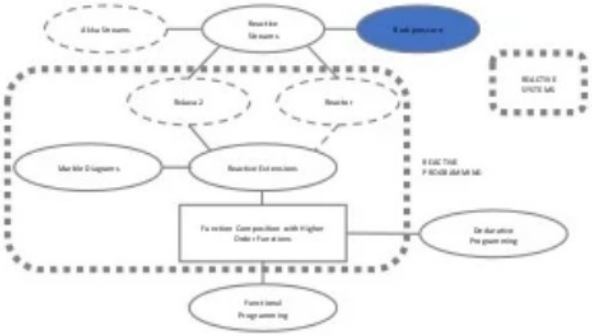
Pulling and Pushing

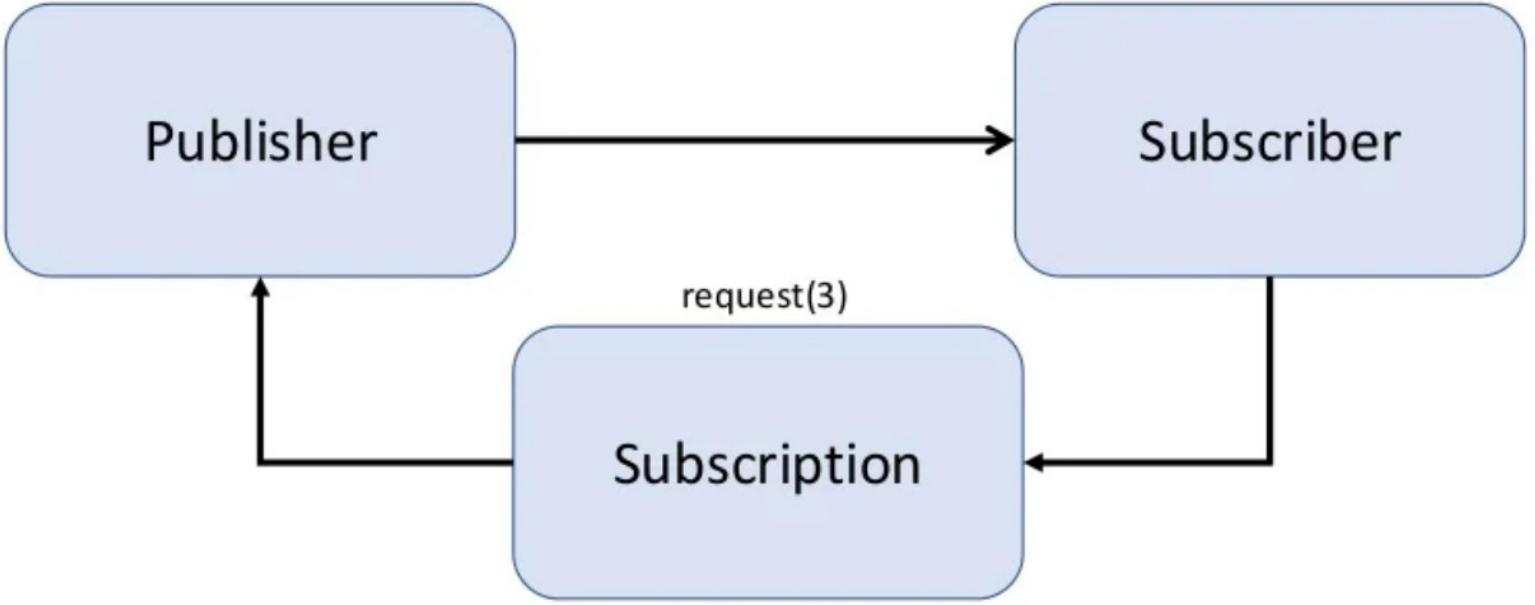




Reactive Programming

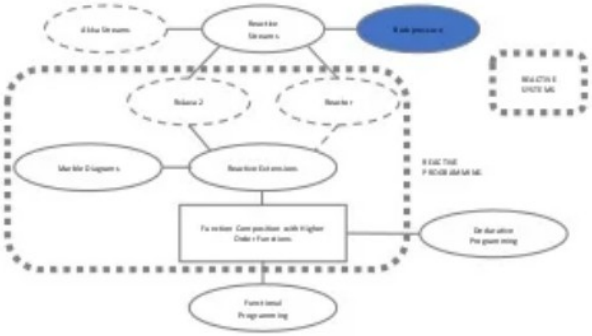
Pulling and Pushing

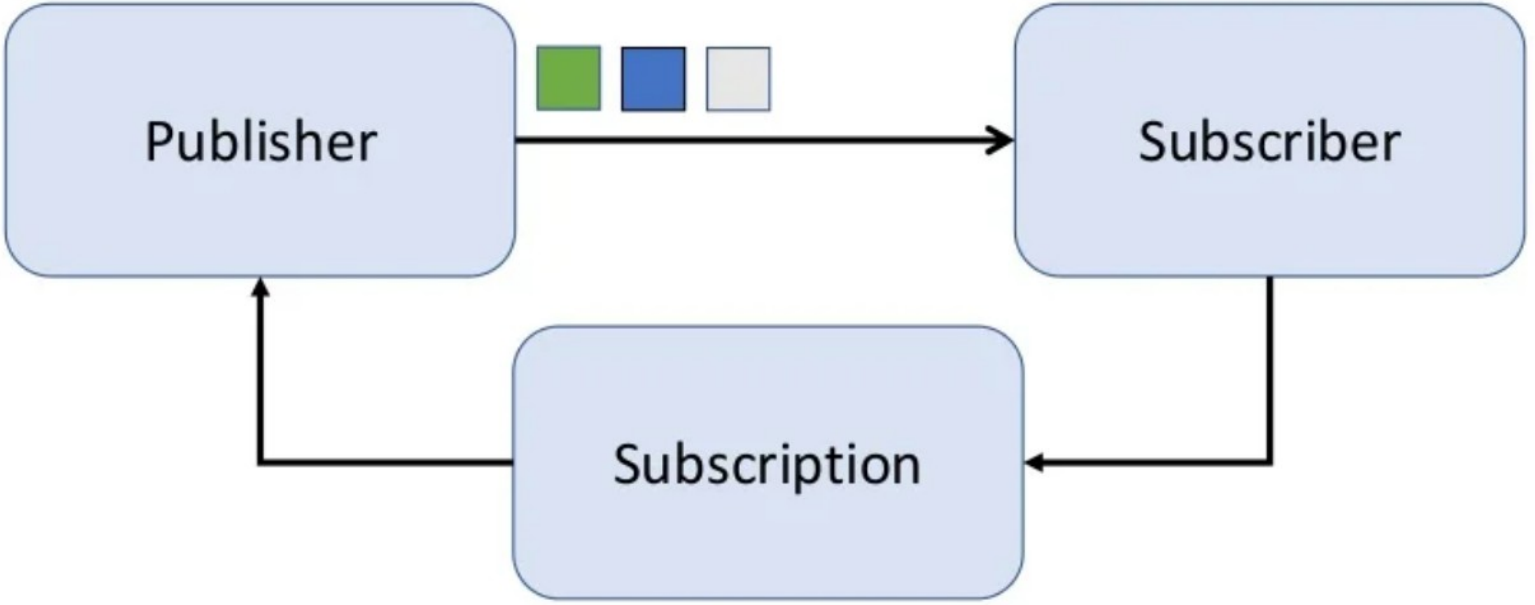




Reactive Programming

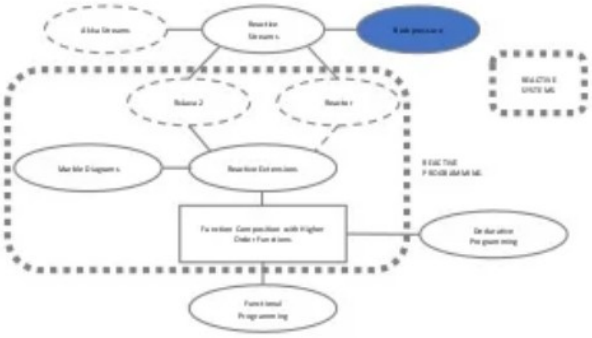
Pulling and Pushing

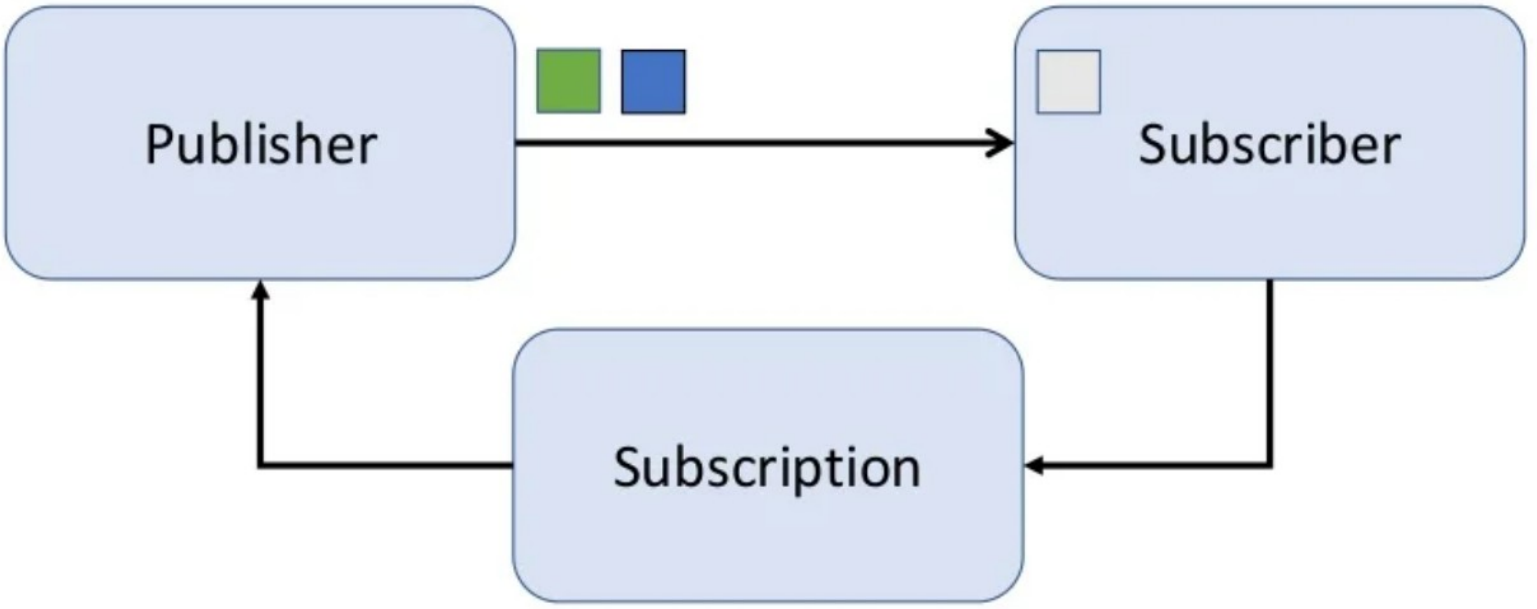




Reactive Programming

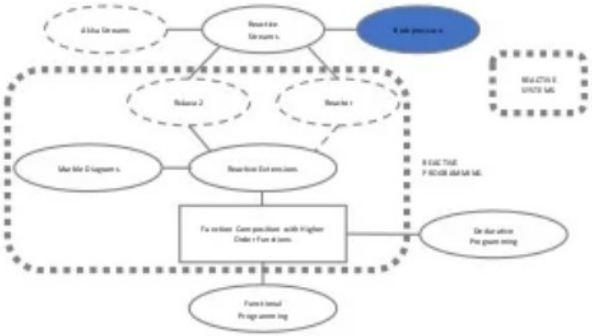
Pulling and Pushing

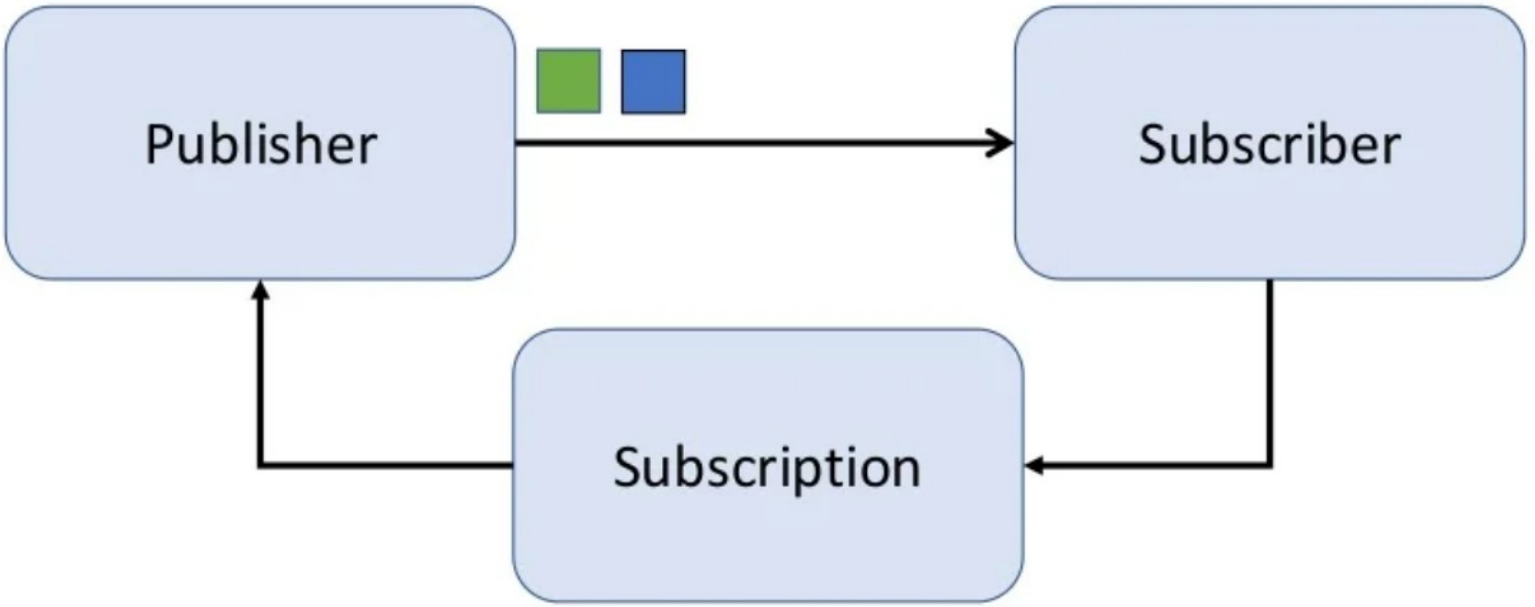




Reactive Programming

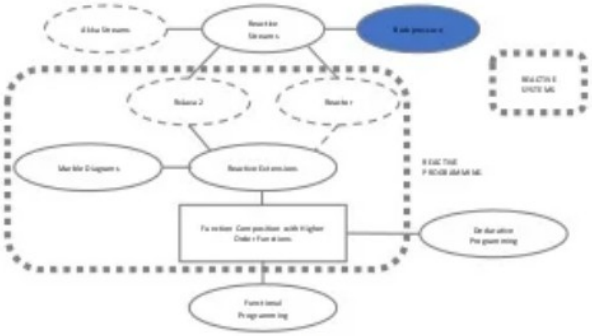
Pulling and Pushing

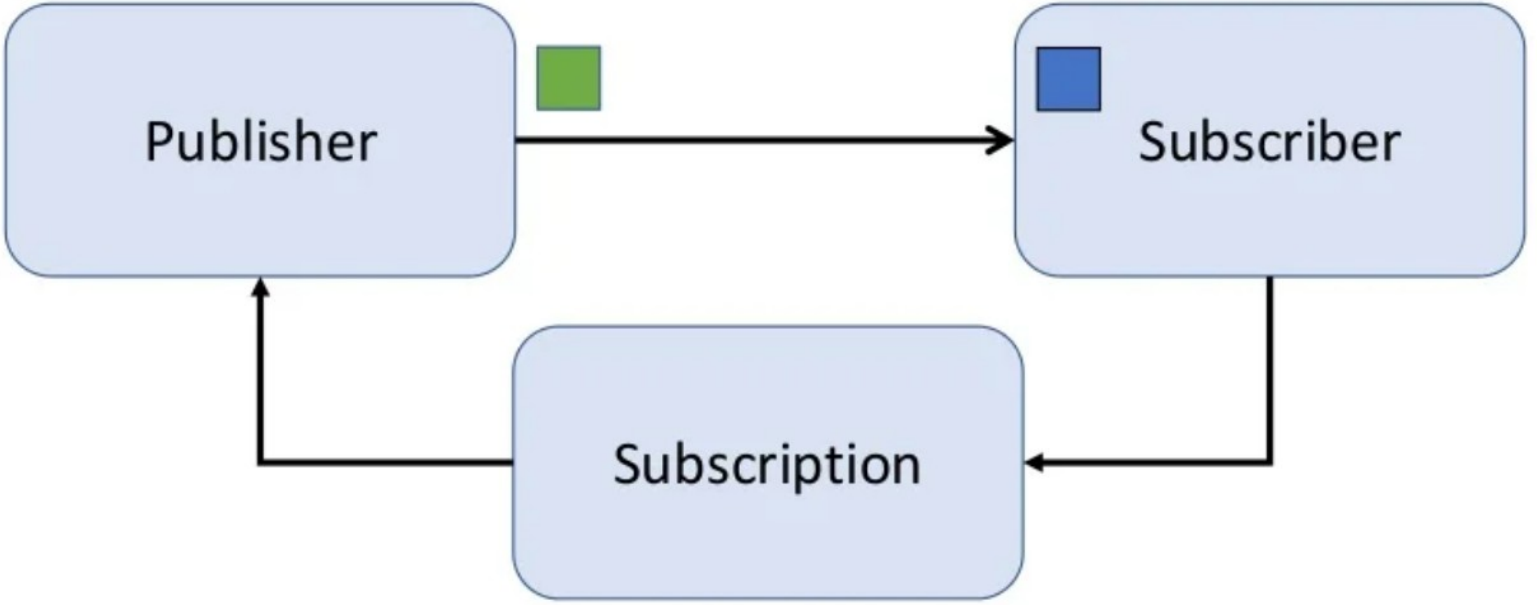




Reactive Programming

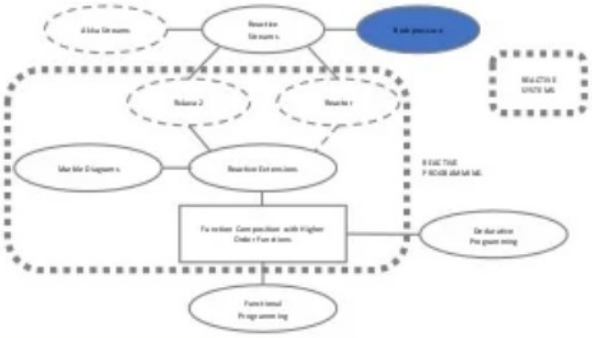
Pulling and Pushing

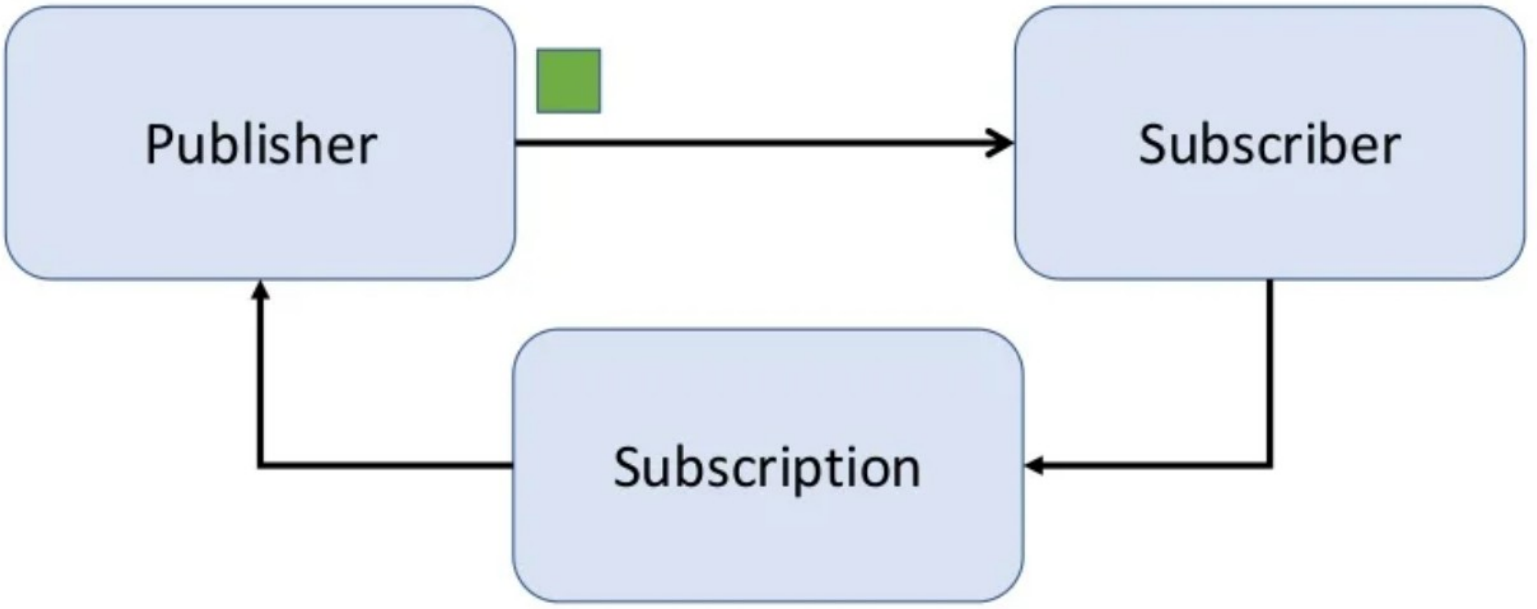




Reactive Programming

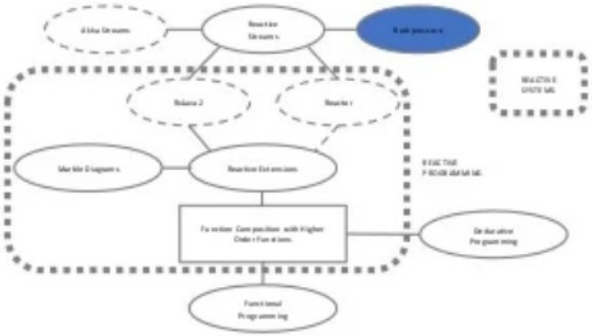
Pulling and Pushing

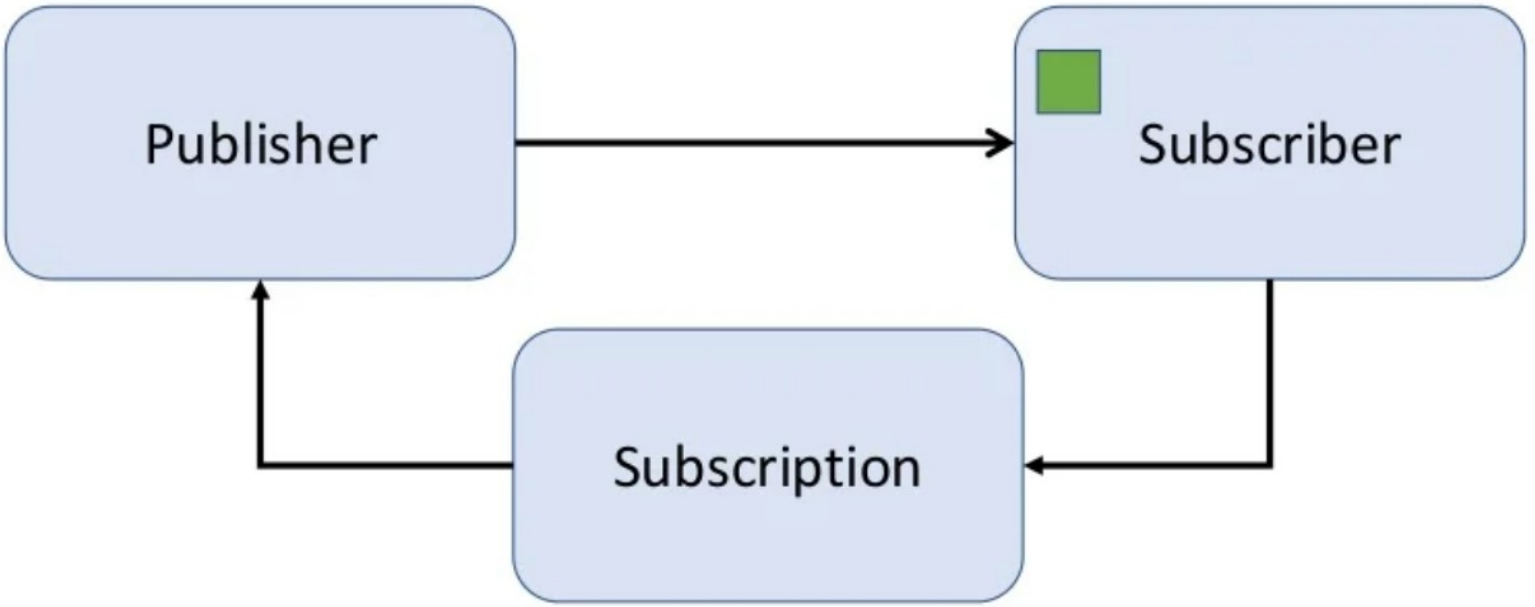




Reactive Programming

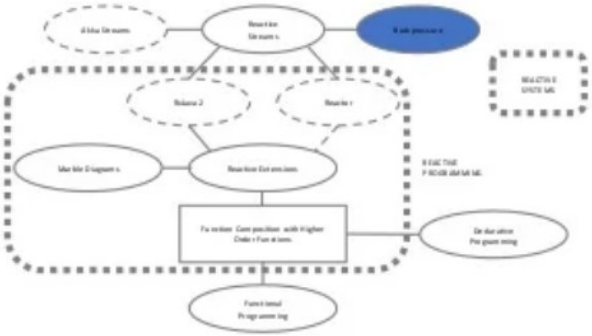
Pulling and Pushing

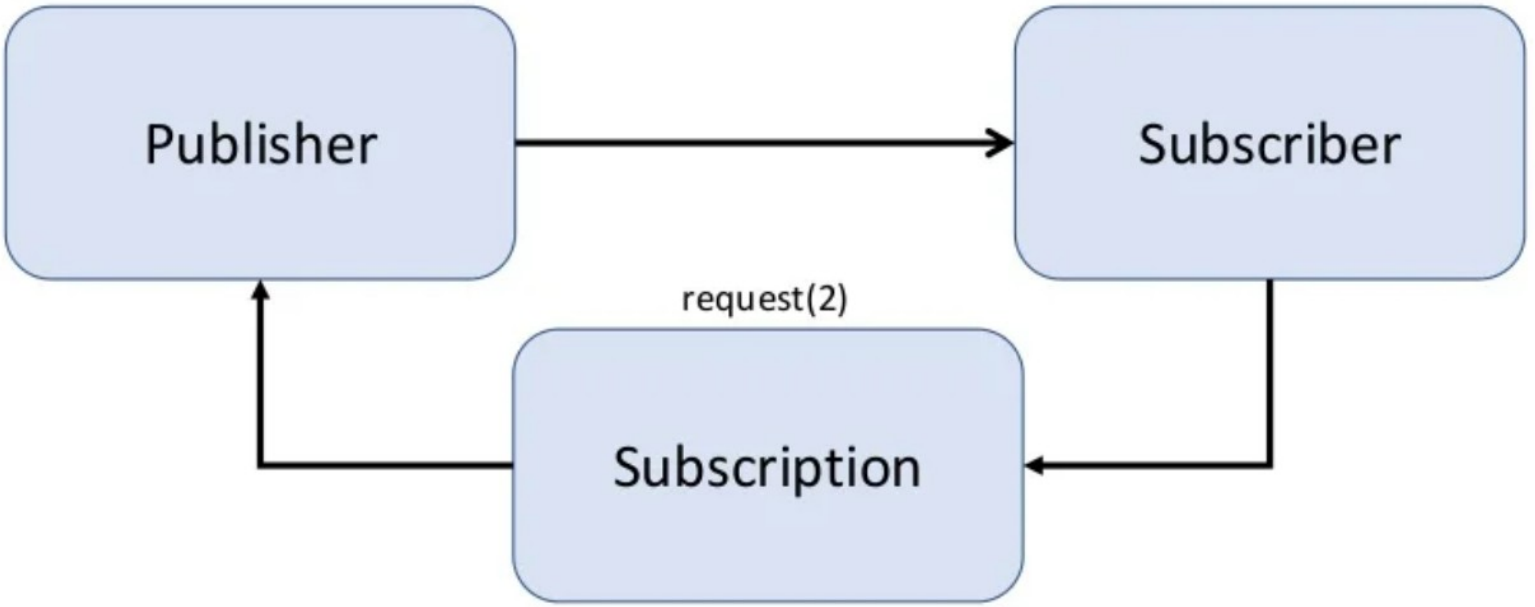




Reactive Programming

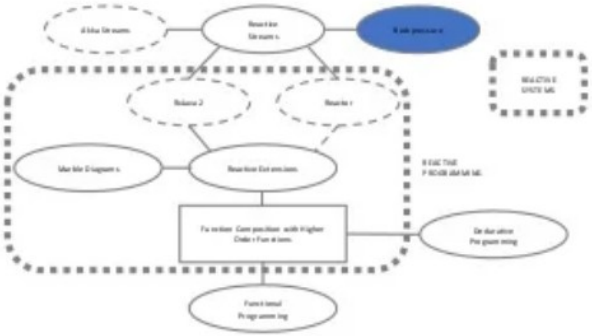
Pulling and Pushing

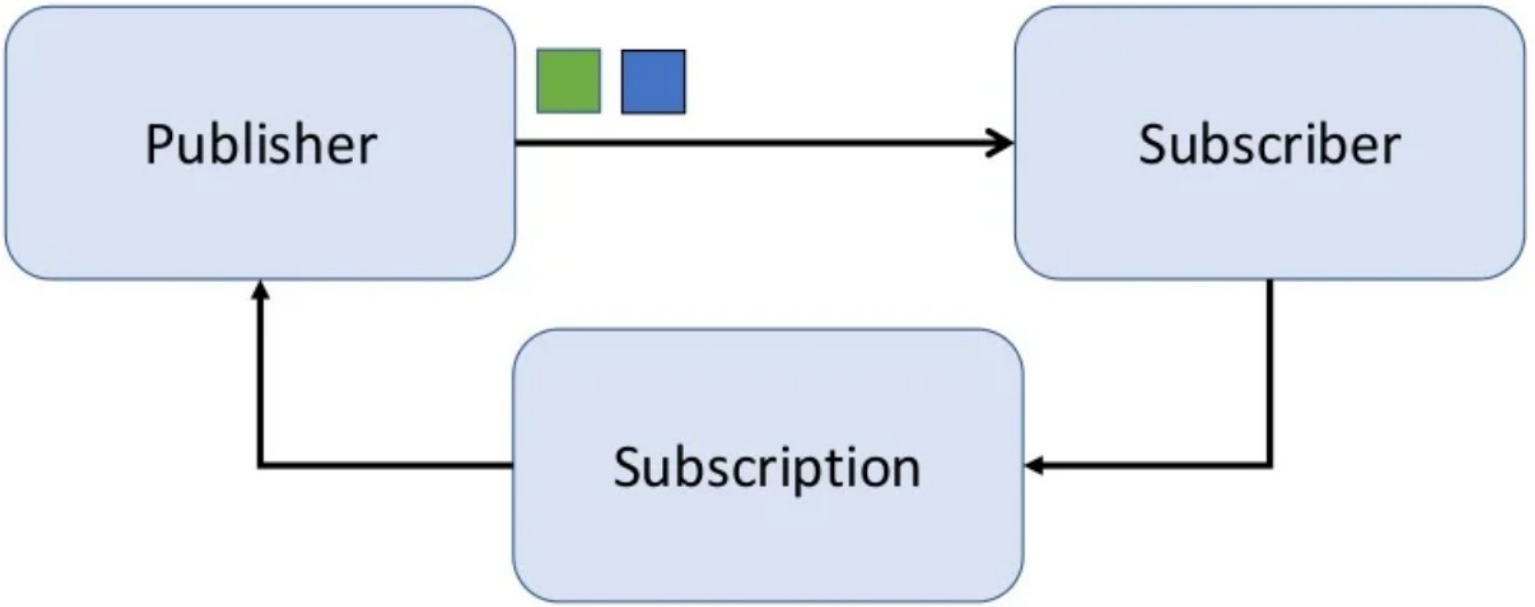




Reactive Programming

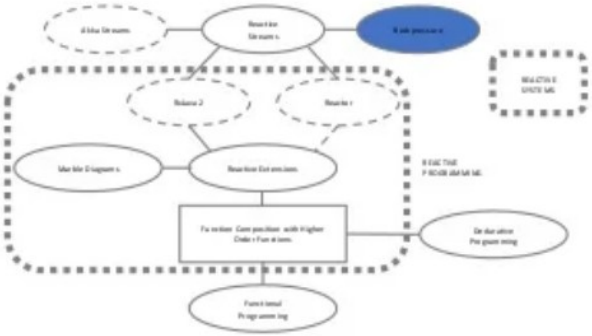
Pulling and Pushing

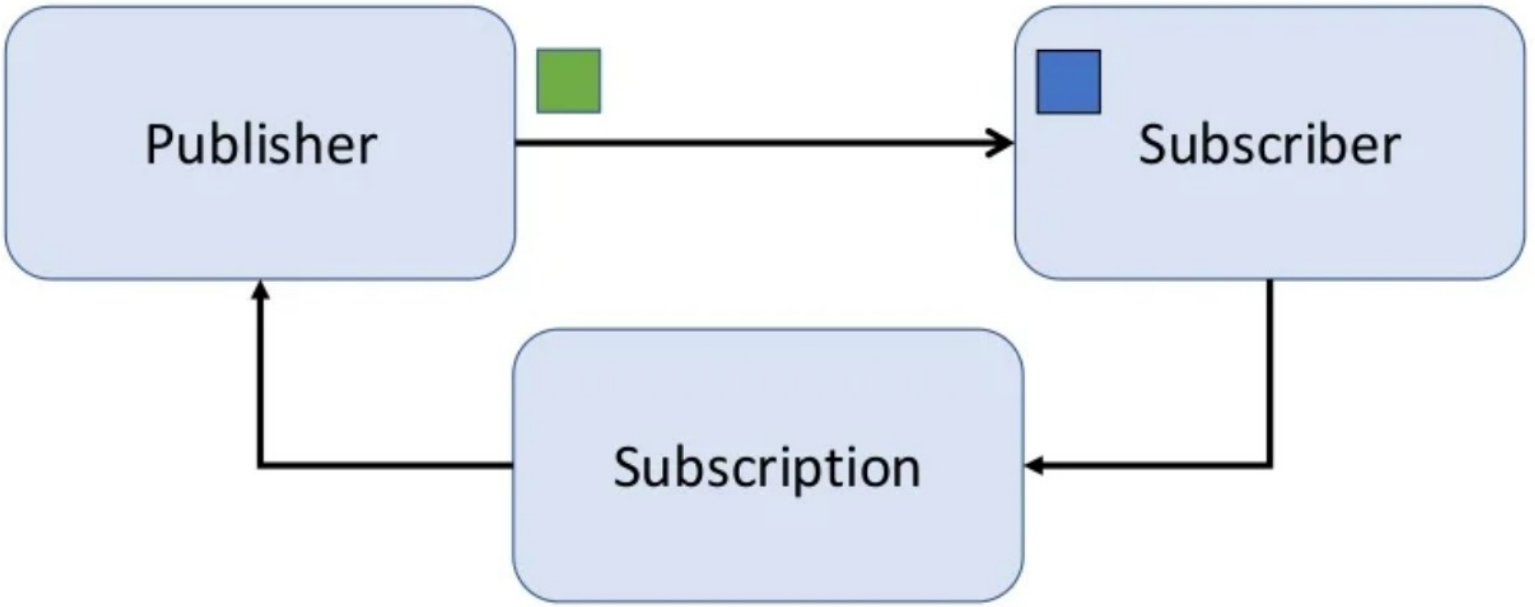




Reactive Programming

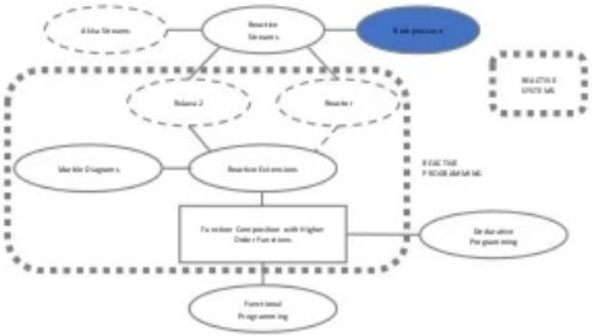
Pulling and Pushing

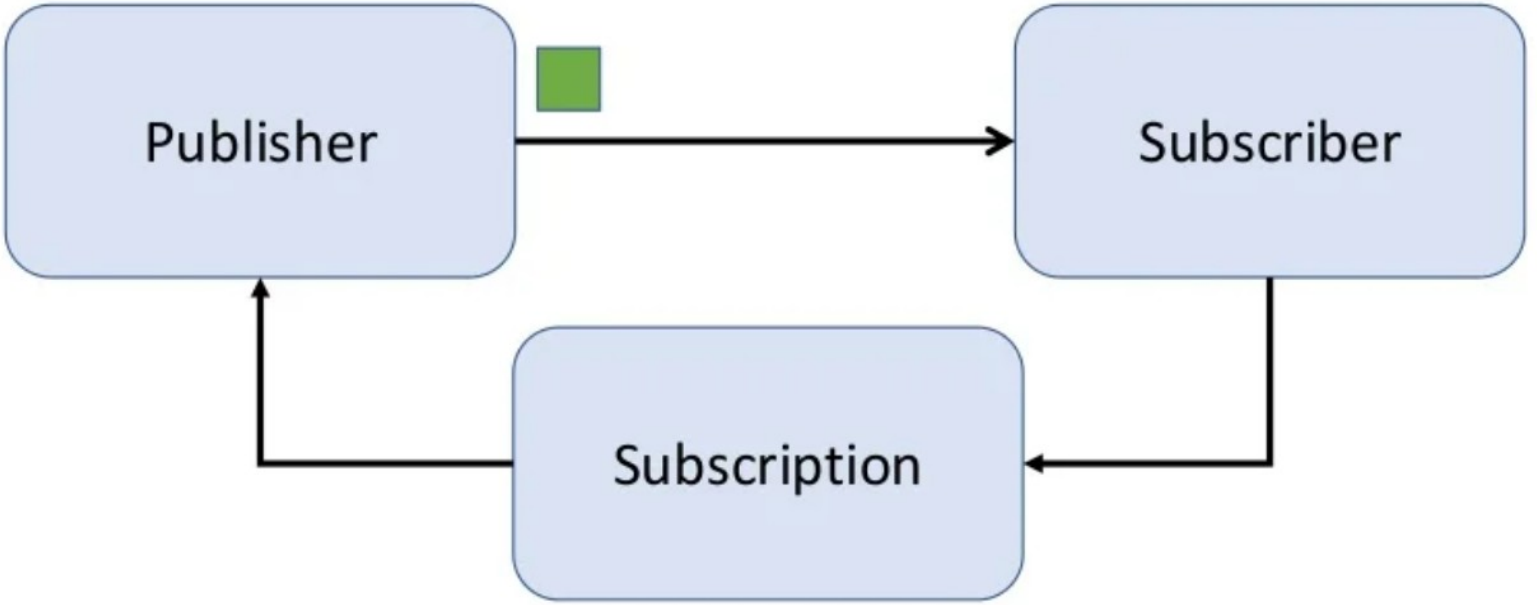




Reactive Programming

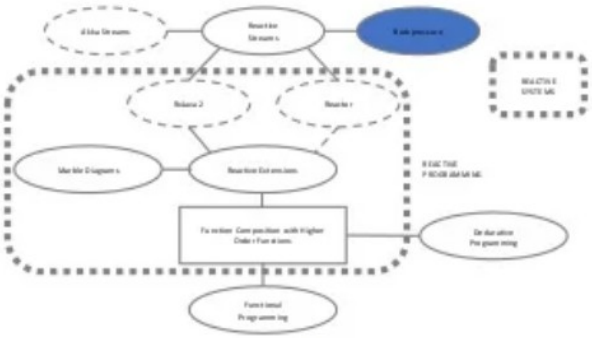
Pulling and Pushing

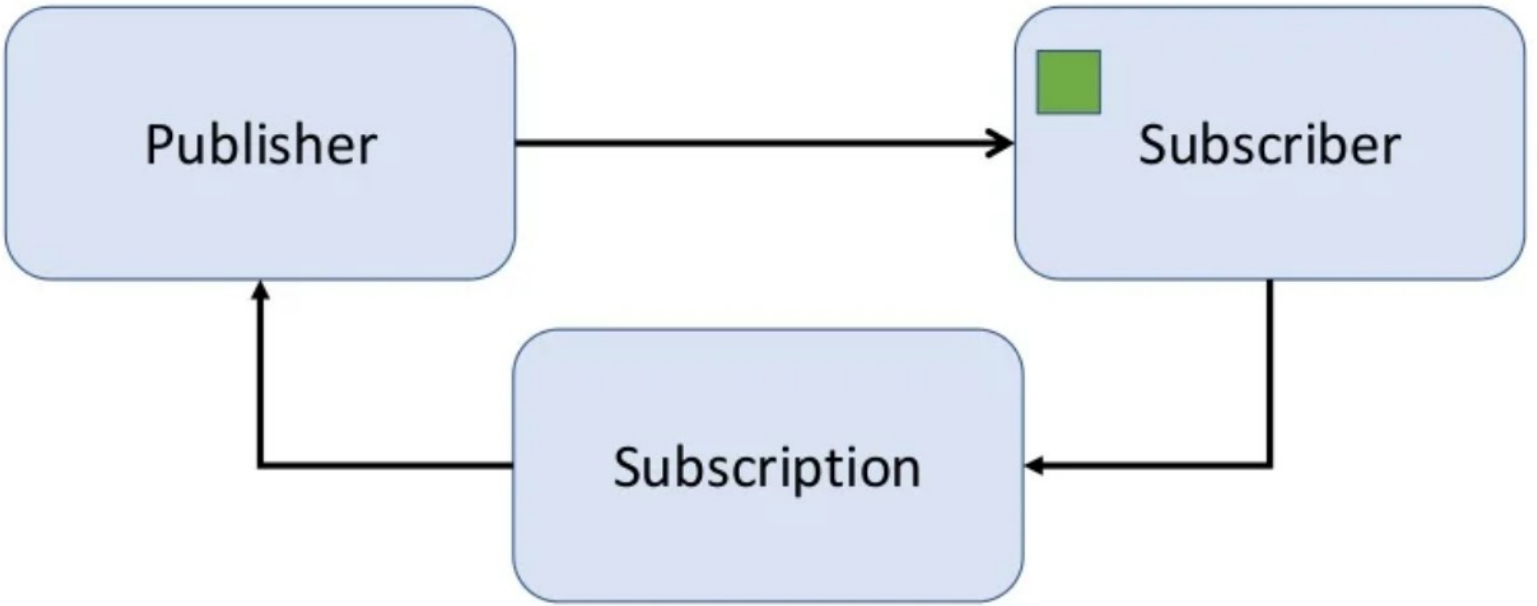




Reactive Programming

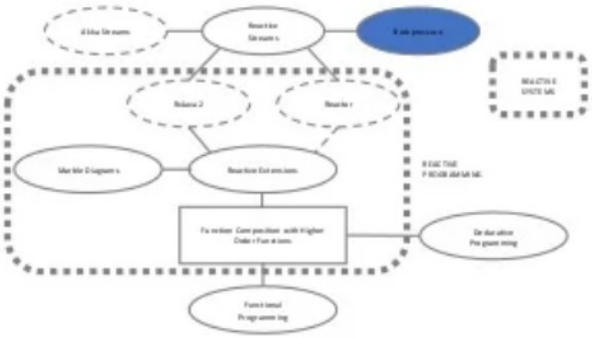
Pulling and Pushing

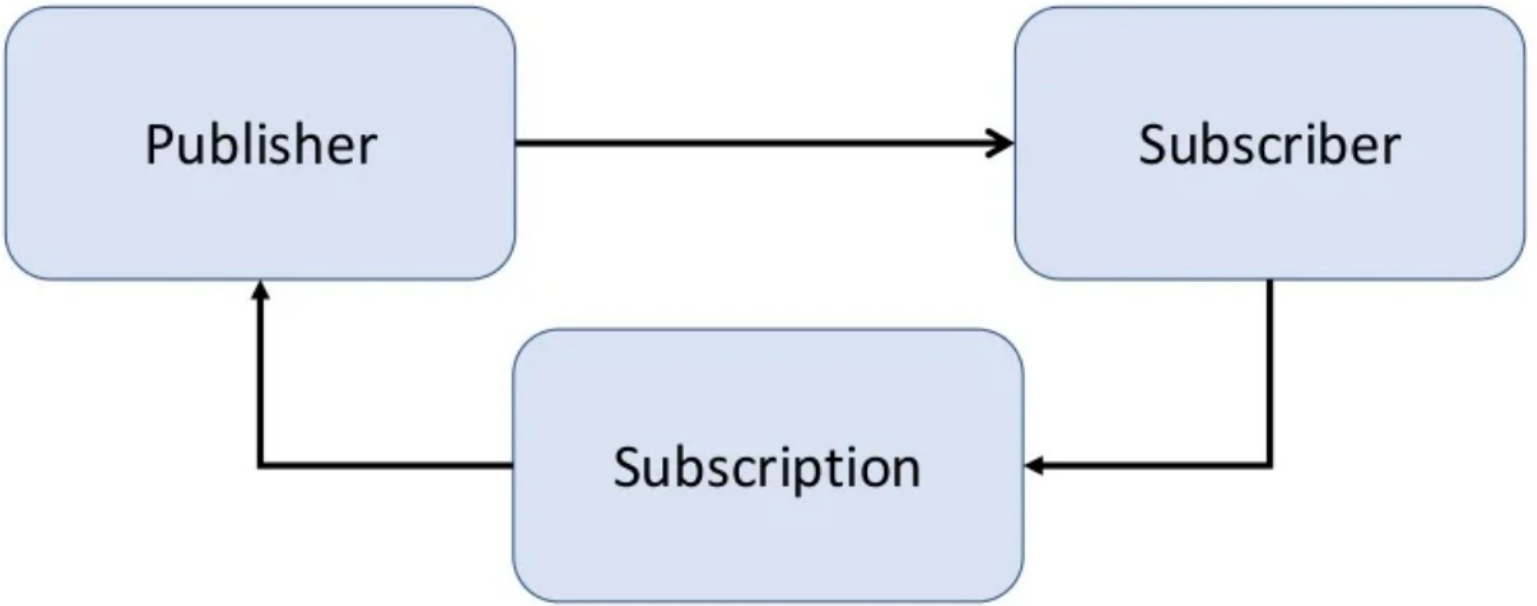




Reactive Programming

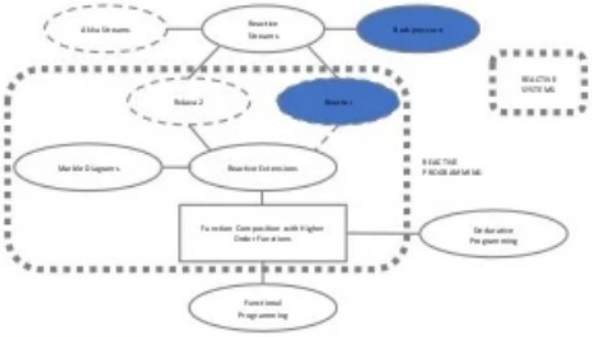
Pulling and Pushing





Reactive Programming

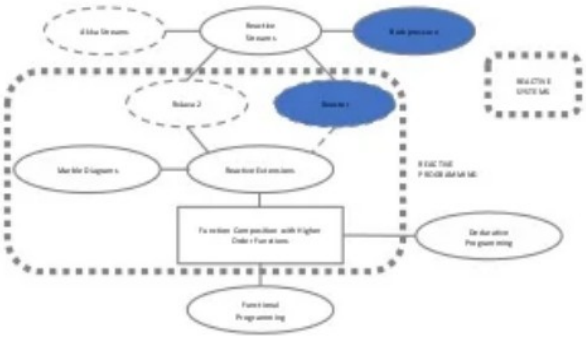
Pulling and Pushing



```
List<User> users = Flux.fromIterable(cwids)
  .flatMap(cwid -> Mono
    .fromCallable(() -> loadUser(cwid))
    .subscribeOn(Schedulers.fromExecutor(executorService)))
  .collectList()
  .block();
```

Reactive Programming

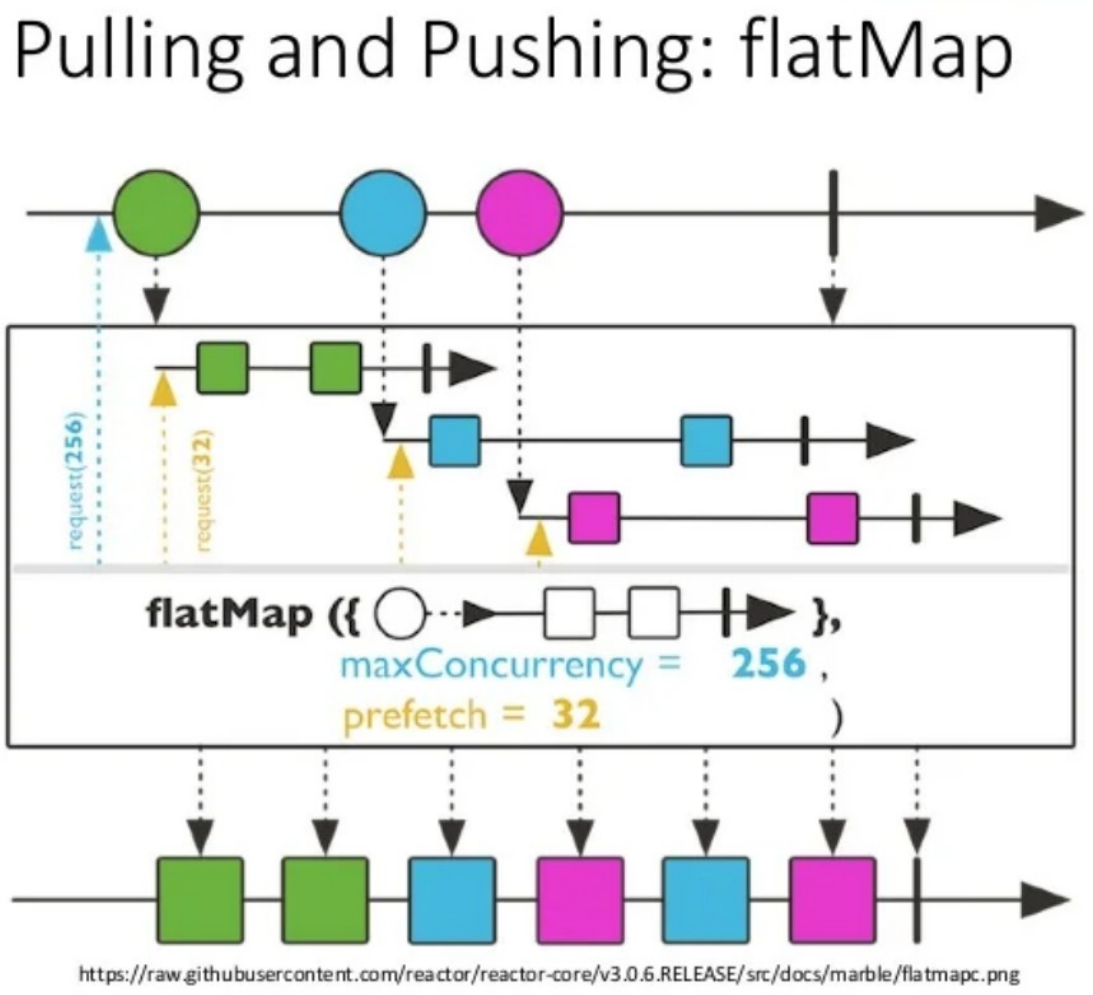
Pulling and Pushing: fromIterable

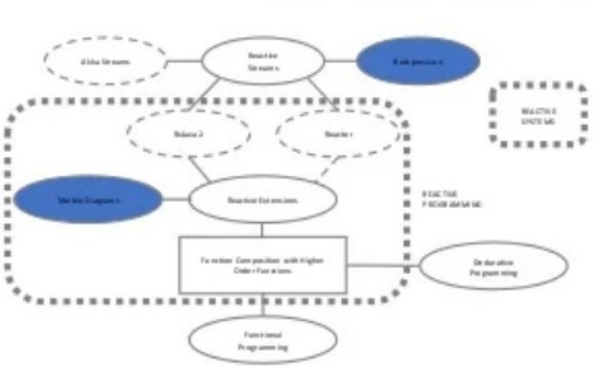


```
class FluxIterable<T> {

  Iterable<? extends T> iterable;

  FluxIterable(Iterable<? extends T> iterable) {
    this.iterable = iterable;
  }

  // more than 500 lines of code

}
```

! naive implementation of `fromIterable` does not enable backpressure

Reactive Programming





Reactive Programming:Operators

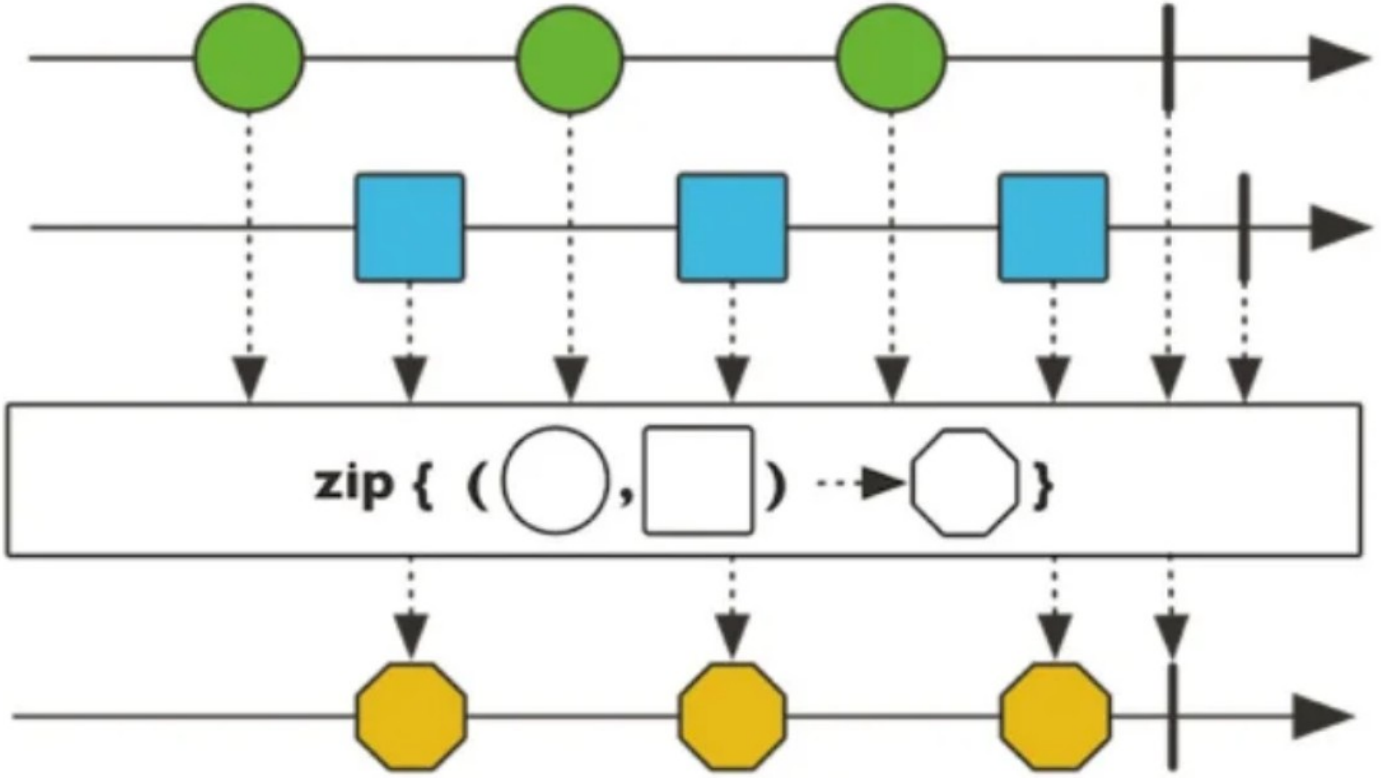
Operators in Reactor

- map / indexed / flatMap / flatMapMany
- collectList / collectMap / count
- concat / merge / zip / when / combineLatest
- repeat / interval
- filter / sample / take / skip
- window / windowWhile / buffer
- ...

https://projectreactor.io/docs/core/release/reference/docs/index.html#which-operator

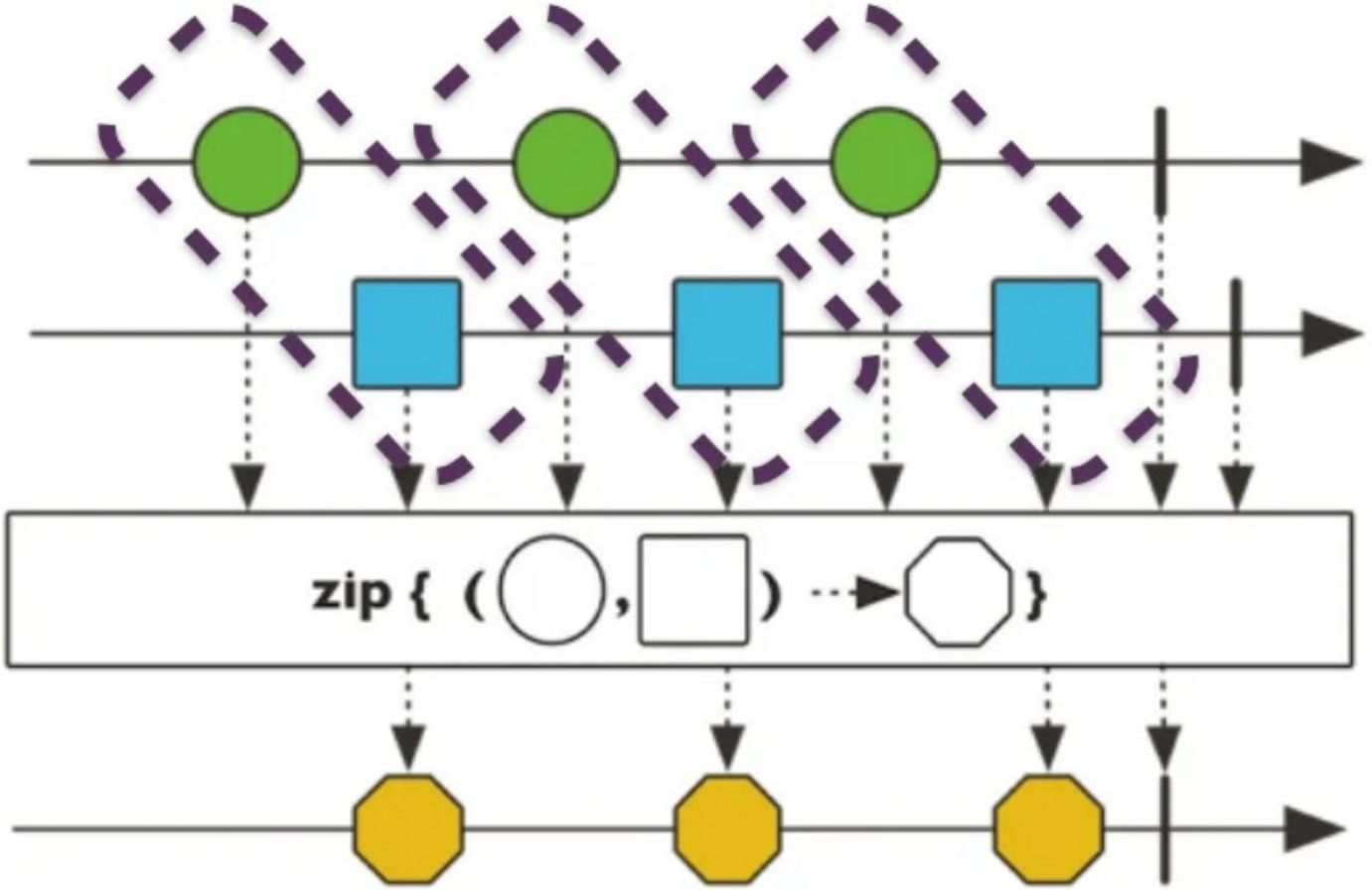
Reactive Programming:Operators

zip



Reactive Programming:Operators

zip



Reactive Programming:Operators

zip

```
Flux<GHUser> github = findGitHubUsers("foo", "bar",
"hoge");
Flux<FBUser> facebook = findFacebookUsers("foo",
"bar", "hoge");

Flux<User> users = Flux.zip(github, facebook)
    .map(tpl -> new User(tpl.getT1(), tpl.getT2()));

users.subscribe();
```